API Security Testing

Bachelorarbeit FS21

Autoren: **Maximilian Lukas Marxer**

E-Mail: maxilian.marxer@ost.ch

Dozentin: Prof. Dr. Nathalie Weiler

Themengebiet: Security / Software

Studiengang: Informatik

Erstellt am: **28. Februar 2021**

Letzte Änderung am: **18. Juni 2021**

Inhalt

Abstract

Aufgabenstellung Das Ziel dieser Arbeit war es, ein Konzept zu entwerfen mit dem man APIs auf ihre Sicherheit testen kann. Dazu wurde eine Beispielapplikation (Spring Boot „Mockup“ API) entwickelt, die den Kauf und Verkauf von Wertpapieren simuliert und eine Zugriffskontrolle implementiert. An der API soll demonstriert werden, wie Schwachstellen durch Sicherheitstests entdeckt werden können. Um die Funktionalität und Sicherheit der API zu überprüfen, sollen neben den üblichen Unit- und Integration-Tests auch End-to-End Tests an der API durchgeführt werden. Für diese Tests soll ein geeignetes Tool in der Arbeit bestimmt werden. Die Tests sollen automatisiert in der CI/CD Pipeline ausgeführt werden.

Methodik Zu Beginn der Arbeit wurde eine Tool-Evaluation über aktuelle Tools zum Testen einer API durchgeführt. Durch die Evaluation wurde ein Tool bestimmt werden, mit welchem die End-to-End Tests an der API durchgeführt werden können. Dabei wurden drei verschiedene Tools genauer untersucht und auf ihre Funktionen analysiert. Anhand der gesammelten Erkenntnisse wurde mit Praxispartner zusammen entschieden, dass die Tests mit dem Postman Tool durchgeführt werden. Nach dieser Entscheidung wurde ein Lösungskonzept entwickelt, um Sicherheitsanforderungen der Beispielapplikation und die dazu nötige Architektur zu definieren. Für die definierte Architektur wurde eine Bedrohungsmodellierung durchgeführt, um die Schwachstellen der Beispielapplikation zu identifizieren und zu priorisieren. Für die Analyse der Schwachstellen wurden die OWASP Top 10 Schwachstellen für APIs verwendet. Nach der Analyse konnten die priorisierten Schwachstellen durch Security API Tests überprüft werden.

Ergebnis Für das Testen der Security wurde eine funktionsfähige API entwickelt. Die Authentifizierung & Autorisierung wurde mit der Keycloak Anwendung umgesetzt und verwendet ein sogenanntes standarisiertes JSON Web Access Token, kurz JWT. Es wurde ein API Gateway aufgesetzt und so konfiguriert, dass alle Anfragen von Benutzern darüber geleitet wird. Er beschränkt die Anzahl der Aufrufe und zeichnet sie auf. Für die Authentifizierung bildet er aus der Signatur des JWT-Tokens einen Opaque-Token. Die Header und Payload Informationen des JWTs bleiben dadurch dem Benutzer verborgen und sind nur dem API Gateway bekannt. Die API wurde mit Unit- und Integration-Tests und End-to-End Tests auf ihre korrekte Funktionalität und Sicherheit getestet. Die Tests wurden automatisiert in die CI/CD Pipeline integriert. Ebenfalls wurde eine statische Code Analyse und ein automatisierter Penetration Test durchgeführt. Die Ergebnisse der Tests haben viele Schwachstellen der Beispielapplikation aufgezeigt. Für einige dieser Schwachstellen wurden Massnahmen getroffen, um diese zu mindern. Die Arbeit zeigt exemplarisch auf, wie eine API auf ihre Sicherheit getestet werden kann.

Management Summary

Ausgangslage

Angesichts von neuen und innovativen Fintech Unternehmungen wie Revolut oder N26 haben sich die Kundebedürfnisse von Bankkunden in den letzten Jahren tiefgreifend verändert. Banken sind gefordert sich weiter zu öffnen und ihren Kunden neue Möglichkeiten mittels APIs anzubieten. APIs werden deshalb auch im Bankensektor vermehrt eingesetzt und müssen den hohen Sicherheitsanforderung einer Bank standhalten. Im Auftrag einer Privatbank soll eine API, mit dem Fokus auf Sicherheit entwickelt werden. An der API sollen fachliche und sicherheitsrelevante Tests durchgeführt werden, um potenzielle Sicherheitsrisiken frühzeitig zu erkennen. Die erstellten Tests sollen nach jeder Änderung im Code automatisiert ausgeführt werden.

Ziele und Vorgehen

Die Arbeit hat das übergeordnete Ziel, ein Konzept zu entwerfen mit dem man APIs auf ihre Sicherheit testen kann. Im Rahmen dieser Arbeit wird dazu eine Beispiel-API entwickelt, die den Kauf und Verkauf von Wertpapieren simuliert und eine Zugriffskontrolle implementiert. Die API wird mit dem Spring Boot Framework in der Sprache Java umgesetzt. Eine Tool-Evaluation für die Umsetzung der Security Tests, sowie die Frage, welche Angriffe und Schwachstellen für APIs existieren, sind ebenso Teil der Arbeit. Das Projekt wurde in einer agilen Vorgehensweise während 17 Wochen und insgesamt 360 Stunden umgesetzt.

Zu Beginn der Arbeit wurden drei aktuelle Tools zum Testen einer API genauer untersucht und miteinander verglichen, um ein Tool für die Sicherheitstest der API zu bestimmen. Das ausgewählte Tool wurde für die Umsetzung der Sicherheitstests verwendet. Anschliessend wurde ein Lösungskonzept entwickelt, um die Sicherheitsanforderung der Beispiel-API zu klären und die dazu notwendige Architektur zu definieren. Das Konzept beinhaltet auch eine Bedrohungsmodellierung, womit potenzielle Schwachstellen des Systems identifiziert und priorisiert wurden.

Ergebnisse

Die geplanten Ziele konnten vollumfänglich umgesetzt werden. Für das Testen der Security wurde eine funktionsfähige REST-API entwickelt. Die API verwendet für die Authentifizierung ihrer Benutzer ein sogenanntes standarisiertes JSON Web Access Token, kurz JWT. Die Zugriffskontrolle verwendet das OAuth2 Protokoll, welches von zahlreichen Unternehmungen als Autorisierungsstandard verwendet ist.

Es wurde ein API Gateway aufgesetzt und so konfiguriert, dass alle Anfragen von Benutzern darüber geleitet wird. Er beschränkt die Anzahl der Aufrufe und zeichnet sie auf. Für die Authentifizierung am Gateway wurde aus Sicherheitsgründen ein Script erstellt, damit nicht das komplette JWT an den Benutzer gesendet wird.

Die API wurde mit Unit- und Integration-Tests und End-to-End Tests auf ihre korrekte Funktionalität und Sicherheit getestet. Die Tests wurden automatisiert in die CI/CD Pipeline integriert. Zusätzlich wurde eine statische Code Analyse und ein automatisierter Penetration Test durchgeführt. Mit den Ergebnissen aus den verschiedenen Tests konnten viele Schwachstellen der API entdeckt werden. Die Arbeit zeigt exemplarisch auf, wie eine API auf ihre Sicherheit getestet werden kann, womit das gesetzte Ziel erreicht worden ist.

1 Technischer Bericht

Gliederung der Arbeit

In diesem Abschnitt soll ein kurzer Überblick über die Gliederung und den Aufbau dieser Arbeit gegeben werden. Nach der Vorstellung des Themengebiets, der Zielsetzung und der Vorgehensweise in dieser Arbeit, werden im Kapitel 1.6 die getätigten Vorarbeiten dieser Bachelorarbeit aufgelistet.

Zu den Vorarbeiten gehören eine Tool-Evaluation, die Definition eines Anwendungsszenarios für die Beispiel-API und ein Überblick über verschiedene API Security Grundlagen. Die Grundlagen werden zum Verständnis der Thematik dieser Arbeit und insbesondere für die im Verlauf dieser Arbeit vorgestellten Bedrohungsmodellierung benötigt.

Im Kapitel 1.7 wird das Lösungskonzept vorgestellt, um Methoden und Ziele für die Sicherheitstests einer API definieren zu können. Dazu beschäftigt sich Kapitel 1.7.1 mit den Sicherheitszielen der API. Aufbauend darauf werden in Kapitel 1.7.2 die Sicherheitsanforderungen der API festgelegt. Kapitel 1.8.3 beschäftigt sich mit der dazu nötigen Architektur. Für die Identifikation und Priorisierung von Sicherheitstests wird in Kapitel 1.7.4 eine Bedrohungsmodellierung vorgestellt.

Kapitel 2 beschäftigt sich mit technischem Aspekt der Softwareentwicklung. Unter Kapitel 2.1 werden die funktionalen und nichtfunktionalen Anforderungen der zu entwickelnden Beispiel-API klar definiert. Kapitel 2.2 stellt die Architektur anhand von UML Diagrammen dar.

Abschliessend werden in Kapitel 3 die Ergebnisse der Arbeit zusammengefasst. Das Kapitel 4 bewertet die Ergebnisse und zeigt Bereiche auf, die nach der Arbeit noch vertieft werden können.

1.1 Ausgangslage

Einleitung

Die Schweizerische Bankiervereinigung (SBVg) versteht unter dem Begriff «Open Banking» [1] ein Geschäftsmodell, das einen standardisierten und sicheren Austausch von Daten zwischen der Bank und Drittparteien beschreibt. Es bezeichnet die Öffnung der Banken durch APIs, welche vertrauenswürdigen Parteien den Zugriff auf Daten ermöglicht und Transaktionen durchzuführen. API ist die Abkürzung für "Application Programming Interface", also eine Schnittstelle, welche die Interaktion einer Applikation mit anderen Applikationen ermöglicht. Jedes Mal, wenn man eine Sofortnachricht mit WhatsApp sendet oder das Wetter auf dem Smartphone abruft, verwendet man eine API.

Angesichts von neuen und innovativen Markteilnehmern haben sich die Kundenbedürfnisse in den letzten Jahren tiefgreifend verändert Banken müssen das Banking für die Kunden vielseitiger und komfortabler gestalten um den Kunden zufriedenzustellen. Gemäss der Schweizerischen Bankiervereinigung ist es angesichts dieser Tatsache keine Frage mehr, ob Open Banking sich etablieren wird, sondern nur noch in welcher Form. [1].

Neben den neuen Kundenbedürfnissen hat sich die globale Finanzindustrie besonders durch eine neue Ära der Transparenz in den letzten Jahren massiv gewandelt. Durch viele neue regulatorische Auflagen und Gesetze wurden sie gezwungen, Daten an die Regulatoren zu senden. Die Europäische Union hat beispielsweise 2018 die überarbeitete Payment Service Directive (PSD2) verabschiedet, welche Banken europaweit verpflichtet, Kundendaten an Regulatoren weiterzugeben. Dadurch sind APIs in den letzten Jahren immer wichtiger für Banken geworden.

APIs bezeichnen traditionell technische Schnittstellen für Systeme. Sie sind heutzutage immer ausgefeilter und machen heute den grössten Teil des Webverkehrs aus. Smartphones werden beispielsweise zum Bezahlen einer QR-Rechnung verwendet, wobei das Gerät Daten über einen API-Aufruf sendet, um die Zahlung des Kunden durchzuführen. APIs machen die Kommunikation zwischen Bank und Kunde schnell, bequem und kostengünstig. Indem sie den Zugang zu gemeinsam genutzten Kundendaten erleichtern, können Kunden ein potenziell besseres Erlebnis bei der Abwicklung ihrer finanziellen Tätigkeiten machen. So könnte eine API beispielsweise die Transkationen analysieren, um für den jeweiligen Kunden ein besseres Angebot zu finden, welches für den Kunden geeignet ist, beispielsweise einen bestimmten Fonds, welcher alle Aktien enthält, die der Kunde bereits einzeln besitzt.

Die Sicherheit und der Schutz einer API ist ein brandaktuelles und wichtiges Thema. Security-Experten sind durch die weit verbreitete Verwendung von APIs besorgt. Gartner [2] schreibt in einem Bericht, dass bis 2022 der API-Missbrauch der häufigste Angriff sein wird, den Security-Experten beobachten. In einer Studie aus dem Jahr 2019 schreibt Gartner [3], dass 40 % der Webanwendungen mehr Angriffsfläche in Form von exponierten APIs als in einem User Interface (UI) anbieten werden. Ausserdem prognostizierte Gartner, dass diese Zahl bis 2021 auf 90 % steigen wird.

Laut einem Bericht, der 2019 von Akamai [4] veröffentlicht wurde, machten API-Aufrufe schon 2018, 83% des Web-Verkehrs aus. Aufgrund des regelrechten Booms von mobilen Apps und Single Page Applikationen kann man davon ausgehen das sich der Anteil noch

vergrössert hat. Aufgrund der genannten Gründe werden im Bankensektor interne und externe APIs immer gefragter. Im Auftrag einer Privatbank soll deshalb ein Konzept erarbeitet werden, welches Methoden aufzeigt, um die Sicherheit und den Schutz einer API zu gewährleisten.

1.2 Vision

Vision

Die Vision der Bachelorarbeit ist es Methoden aufzuzeigen, wie APIs auf ihre Sicherheit getestet werden können. Angriffe auf APIs werden immer populärer und deshalb steigt auch die Nachfrage nach ihrer Sicherheit. Die zu entwickelnde Spring Boot API soll demonstrative Schwachstellen enthalten, um zu prüfen, ob diese durch Tests entdeckt werden können.

1.3 Zielsetzung

Spring Boot API

Das Ziel dieser Bachelorarbeit ist es, ein Konzept zu entwerfen mit dem man APIs auf ihre Sicherheit testen kann. Dazu soll eine Beispielapplikation (Spring Boot „Mockup“ API) entwickelt werden, die den Kauf und Verkauf von Wertpapieren simuliert und eine Zugriffskontrolle implementiert.

Tool-Evaluation für Security Tests

An der API soll demonstriert werden, wie Schwachstellen durch Sicherheitstests entdeckt werden können. Um die Funktionalität und Sicherheit der API zu überprüfen, sollen neben den üblichen Unit- und Integration-Tests auch End-to-End Tests an der API durchgeführt werden. Für diese Tests soll ein geeignetes Tool in der Arbeit evaluiert werden. Die Tests sollen automatisiert in der CI/CD Pipeline ausgeführt werden.

Security Testing

Für das Testen der Security soll ein Lösungskonzept entworfen werden, um die Sicherheitsanforderungen der Beispielsapplikation zu erfüllen. In Absprache mit dem Praxispartner wird die zu testende Architektur definiert und auf ihre Schwachstellen untersucht. Da nicht alle Schwachstellen getestet werden, soll geprüft werden welche Schwachstellen testbar sind und eine Priorisierung vorgenommen werden. Die priorisierten Schwachstellen wurden mit Postman getestet.

1.4 Abgrenzung

Technologie

Der Praxispartner hat vorgegeben ein Backend mit einem Spring Boot Service zu entwickeln, welches eine API zur Verfügung stellt. Der API verwendet als Datenbankmanagementsystem Postgres. Für die Authentifizierung und Autorisierung wurde vom Praxispartner vorgeben das OAuth2 Protokoll in Verbindung mit dem Open ID Connect (OIDC) Protokoll zu verwenden. Dazu soll die frei verfügbare Software „Keycloak“ verwendet werden, da sie auf den geforderten Protokollen aufbaut und diese unterstützt.

Spring Framework und Spring Boot

Spring hat sich als vertrauenswürdiges Framework für die Entwicklung von Applikationen in der Java Welt bewiesen und ist weit verbreitet. Es ist nicht mehr gültig, Spring als ein Framework zu bezeichnen, da es eher ein Überbegriff ist, der verschiedene Frameworks umfasst. Eines dieser Frameworks ist Spring Boot, welches die Entwicklung von Spring-Anwendungen stark vereinfacht. Die Autokonfiguration und Starter-Abhängigkeiten reduzieren die Menge an Code und Konfiguration, die benötigt wird, um eine API zu entwickeln und starten. Das Hauptziel des Spring Boot-Frameworks ist es, die Gesamtentwicklungszeit zu reduzieren und die Effizienz zu erhöhen, indem es eine Standardkonfiguration für Unit- und Integrationstests bietet. [5]

Der Praxispartner verwendet dieses Framework für die Entwicklung von APIs und es wurde vorgeben dieses in der Arbeit zu verwenden.

Systemlandschaft

Die Entwicklung das Deployment der Applikation findet ausserhalb der internen Systemlandschaft statt, weshalb dies auch nicht Teil der Arbeit ist.

Es wurde vereinbart das die API und weitere Microservices auf dem privaten Gerät entwickelt wird. Für das Deployment wurde die Cloud Plattform «Heroku» ausgewählt. Lokal werden Docker-Container verwendet, welche es später vereinfachen die Microservices in der internen Systemumgebung zu deployen.

1.5 Methodik

Startphase

Der Projektplan wurde in Anlehnung an die *Rational Unified Process*-Methode [6] entwickelt. Zu Beginn der Arbeit wurde eine Tool-Evaluation über aktuelle Tools zum Testen einer API durchgeführt. Dadurch sollen die Stärken und Schwächen der einzelnen Tools analysiert und gegenübergestellt werden, um ein geeignetes Tool für die Sicherheitstests der API zu bestimmen. Dazu wurden drei verschiedene Tools ausgewählt und tiefer untersucht. Anhand der gesammelten Erkenntnisse wurde mit Praxispartner zusammen entschieden, dass die Tests mit dem Postman Tool durchgeführt werden.

Für die Definition von Sicherheitstests wurde entschieden eine Bedrohungsmodellierung in der späteren Konzepthase durchzuführen. Es wurde deshalb in Absprache mit dem Praxispartner ein Anwendungsszenario für die zu entwickelnde API festgelegt, um in einem ersten Schritt den Zweck und die Datenbasis für die API zu klären. Nachdem der Zweck der API geklärt wurde, stand an nächster Stelle eine Analyse der API Security Grundlagen. Das Ziel der Analyse war es, einen Überblick über das Risikoprofil einer API, für die spätere Bedrohungsmodellierung in der Konzeptphase zu erhalten.

Konzeptphase

Nach der Startphase wurde ein Konzept für die Entwicklung einer Spring-Boot API erstellt. Es wurden UML-Entwürfe und Use-Case Diagramme erstellt, welche mit dem Betreuer diskutiert und im Anschluss verbessert und erweitert wurden. Es wurden Ziele für die API definiert, um daraus Sicherheitsanforderungen zu erstellen. Um die getroffenen Sicherheitsanforderungen erfüllen zu können wurde in Absprache mit dem Praxispartner eine Architektur definiert. Für die geplante Architektur wurde eine Bedrohungsmodellierung erstellt, um die Schwachstellen der Beispielapplikation zu identifizieren und zu priorisieren. Für die Analyse der Schwachstellen wurde die Top 10 Sicherheitsrisiken für APIs verwendet, welche vom Open Web Application Security Project (OWASP) Ende 2019 veröffentlicht wurde.

Entwicklungsphase

Nachdem das Konzept ausgearbeitet war, begann die Entwicklung der Spring API. Da noch keine Erfahrung mit Spring vorhanden waren, musste zuerst eine Einarbeitung in die Spring API erfolgen. Der Leitfaden für die Entwicklung der API war das erstelle Domainmodell mit den Use Cases. Anschliessend wurde mit der Software „Keycloak“ und dem Spring Security Framework die Authentifizierung und Autorisierung für die API umgesetzt. Ein API Gateway wurde konfiguriert, um alle Anfragen von Benutzern darüber zu leiten, damit sie aufgezeichnet und die Anfragen limitiert werden können. Für die Authentifizierung am Gateway wurde ein Javascript erstellt, welches aus der Signatur des JWT-Tokens einen Opaque-Token bildet. Die Header und Payload Informationen des JWTs bleiben dadurch dem Benutzer verborgen und sind nur dem API Gateway bekannt. Die API wurde mit Unit- und Integration-Tests und End-to-End Tests auf ihre korrekte Funktionalität und Sicherheit getestet. Die Tests wurden automatisiert in die CI/CD Pipeline integriert. Ebenfalls wurde eine statische Code Analyse und ein automatisierter Penetration Test durchgeführt.

Abschlussphase

In der Abschlussphase werden die Ergebnisse dieser Arbeit zusammengefasst und aufgezeigt und bewertet. Als Ausblick werden Bereiche aufgezeigt, welche nach der Arbeit noch vertieft werden könnten.

1.6 Vorarbeiten

Inhalt

In dem folgenden Kapitel werden die Resultate der getätigten Vorarbeiten beschrieben-. Das Kapitel beinhaltet die nachfolgenden drei Unterkapitel:

- Tool-Evaluation für automatisierte Security Tests
- Szenario der API klären
- API Security Grundlagen

Die Resultate der einzelnen Kapitel waren die Grundlage für die wichtigsten Entscheidungen für das nachfolgende Lösungskonzept, welches im Kapitel 1.7 beschrieben wird.

1.6.1 Tool-Evaluation für automatisierte Security Tests

Einleitung

In diesem Kapitel wird die Tool-Evaluation für die Umsetzung von automatischen Security Tests für die entwickelte Prototyp API beschrieben. Bei der Verwendung eines API-Testtools nutzen QAs in der Regel entweder die Vorteile der fixfertigen Lösungen oder sie entwickeln ein individuelles Framework aus den Komponenten des Tools. Es werden deshalb zwei fixfertige Lösungen und ein individuelles Framework miteinander verglichen.

Kandidaten

Es gibt eine riesige Auswahl von API-Testing Tools. In dieser Evaluation beschränken wir uns auf die nachfolgenden drei Tools:

Katalon Studio

Katalon ist ein kostenloses Testautomatisierungstool für API, Web, Desktop App und Mobile Anwendungen. Gemäss der eigenen Webseite benutzen schon über 65'000 Unternehmungen darunter auch viele namenhafte wie ORACLE, Sony oder SAP. Es entwickelt sich zu einem führenden Tool für API/Web-Services-Tests und positioniert sich als umfassende End-to-End-A)utomatisierungslösung für Entwickler und Tester. (vgl. [7])

Postman

Postman ist ein weitverbreites und bekanntes Tool. Die meisten Entwickler denekn, dass Postman nur für manuelle explorative Tests geeignet ist. Es wurde in der Vergangenheit häufig benutzt um die Funktionalität einer API mittels manuellen Tests zu validieren.

Postman kann als Browser-Erweiterung oder als Desktop-Anwendung auf Mac, Linux und Windows installiert werden. Es wird von einigen Automatisierungstestern für API-Tests verwendet und hat sich als eine Umgebung zum Entwickeln und Testen von APIs entwickelt. [8]

Rest Assured

REST Assured ist eine Java-Bibliothek zum Erstellen eines individuellen API Testframework (vgl. [9]). Es wurde speziell für das Testen entwickelt und lässt sich in jedes bestehende Java-basierte Automatisierungsframework integrieren. Es bietet eine BDD-ähnliche Sprache an, die das Erstellen von API-Tests in Java sehr einfach macht. Rest-Assured lässt sich beispielsweise auch mit dem Serenity-Automatisierungsframework verwenden, was bedeutet, dass UI- und Rest-Tests in einem Framework kombinieren können, wie dies bei Katalon der Fall ist (vgl. [10])

Kriterien

Für die Evaluierung werden die folgenden Kriterien für die Bewertung des Tools verwendet:

- Funktionalität und Aufwand
- Dokumentation und Support
- Usability
- Wiederverwendbarkeit
- Benötigte Skills

Funktionalität

An das entsprechende Testing Tool werden einige funktionalle Anforderungen gestellt, welche zwingend erfüllt werden müssen. Ein Test-Cases durchläuft typischerweise die nachfolgenden Phasen:

- Authentifizierung: Das Tool muss sich mittels OAuth2 und OIDC authentifzieren können. Es wird die Implementierung von Untestützenden Werkzeugen bewertet.
- Erstellen eines Testfalls (Vor und Nachbedingungen, Beschreibung, Logische Trennung von Testfällen).
- Durchführen eines Testfalls (Bilden von Testdurchführungen, Automatisierbarkeit). Die Tests, welche im Tool erstellt werden müssen automatisierbar sein und in die Gitlab CI/CD eingebaut werden können.
- Auswertung eines Testfalls. Wie wird der Report generiert ? Wie können Testdurchläufe verglichen werden ?

Bei der Bewertung der Funktionalität werden diese Phasen durchlaufen und bewertet.

Dokumentation und Community Support

Ausführliche und verständliche Dokumentationen und eine aktive Online-Community erleichtern die Implementierung und den nachfolgenden Betrieb massgebend. Fehlende oder unklare Dokumentationen erhöhen den Aufwand eines Entwicklers.

Usability

Die Benutzeroberfläche des Tools wird bewertet. Eine gute User Experience verringert den Aufwand um zukünftige Mitarbeiter in das Tool einzuarbeiten.

Wiederverwendbarkeit von Code

Es wird bewertet wie gut die Wiederverwendbarkeit von geschriebenen Code ist. Beispielweise ob der Code für die Authentifzierung dupliziert werden muss oder zentral ausgelagert werden kann.

Benötigte Skills

Welche Programmiersprachen werden unterstützt. Wie viel Programmierkenntnisse sind nötig für das Erstellen eines Testfalls.

Bewertung

Für die Bewertung der Tools wird der nachfolgende Bewertung Schlüssel verwendet:

Bewertung	++	+	o	-
Beschreibung	hervorragend	gut	mässig	schlecht

Testfälle

Für die Tool-Evaluation wurden die nachfolgenden Testfälle definiert:

- Testen eines geschützen Endpoints: Um einen ersten Eindruck über das Tool zu erlangen, wird ein Test für einen geschützen Endpoint durchgeführt Es muss dafür zuerst das Access Token vom Auth-Service bezogen werden.
- Reporting Funktionen: Es soll geprüft werden wie Berichte generiert werden können. Diese Fähigkeit ist von entscheidender Bedeutung, Unternehmen einen robusten, datengesteuerten Ansatz bei der Entscheidungsfindung zu bieten.
- Einbau in die CI/CD Pipeline

1.6.1.1 Katalon Studio

Testen eines geschützten Endpoints

Katalon wirbt damit, dass es einfach sei Test zu schreiben und keine Coding-Skills benötigt werden um einen Test zu erstellen. Beim Testen eines geschützen Endpoints konnte dieses Versprechen nicht gehalten werden. Katalon Studio bietet für den Bezug eines JWT Access Tokens ein UI an. Die Anmeldeinformationen können wie in der Abbildung eingeben werden. Bei erfolgreicher Authentifizierung erhält man ein Access Tokens, welches durch einen Klick auf „Update to http Header“ zum Header des jeweiligen Request hinzugefügt wird. Dieser Button hat sich auch zum Problem beim Laufen des Tests gezeigt. Als das Access Token nicht mehr gültig war, wurde nicht automatisch ein neues Token bezogen und in den Header hinzugefügt. Es musste immer manuell durch Klicken eines Buttons das Token neu bezogen werden und nochmals dasselbe für das Hinzufügen in den Header des Requests.

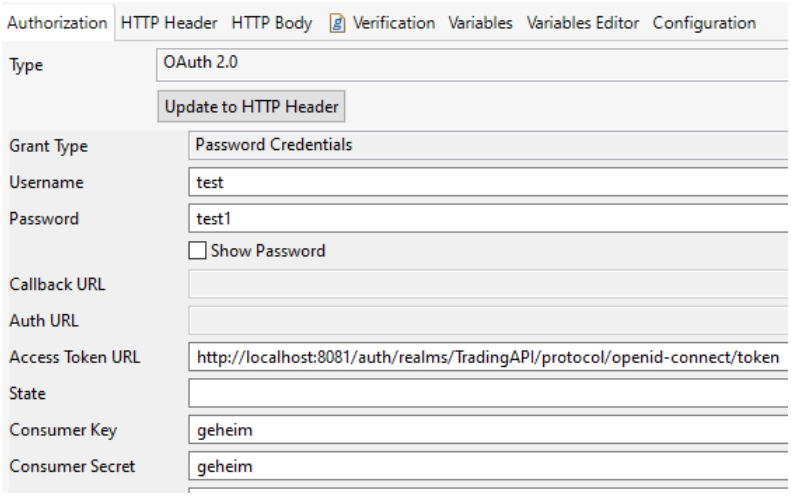


Abbildung 1: Authentifizierungs-UI von Katalon

Weiter negativ aufgefallen ist, dass die hinterlegen Informationen wie in der Abbildung zu sehen beim Beenden von Katalon immer wieder gelöscht werden. So ist die Funktion für automatische Tests nicht zu gebrauchen. Im Katalon Forum wird bestätigt das die Authorisierungs-Informationen aktuell nicht gespeichert werden können und dies für die Zukunft angedacht ist. (vgl. [11]).

Angesichts der obigen Erkenntnisse musste die Authentifizierung mit Groovy / Java umgesetzt werden. Das Schreiben eines Groovy-Scripts in Katalon für die Authentifizerung war schwieriger als Anfangs angenommen. Katalon bietet für dieses Thema keine Dokumentation an und die Fragen im Forum sind meist unbeantwortet.

Das Erstellen des Scripts hat viel Zeit in Anspruch genommen, da noch keine Erfahrungen mit Groovy bestanden und auch weil die Benutzeroberfläche von Katalon gewöhnungsbedürftig ist. Das UI von Katalon bietet sogenannte Built-In-Keywords, um fixfertige Funktionen zu verwenden. Durch diese wird im Hintergrund automatisch der zugehörige Code generiert. Das erstellen eines Testcase mit solchen Keywords hat sich als sehr umständlich herausgestellt. Für eine Zeile Code müssen dutzende Objekte ausgewählt und Parameter abgefüllt werden. Meiner Meinung ist das UI nicht benutzerfreundlich aufgebaut.

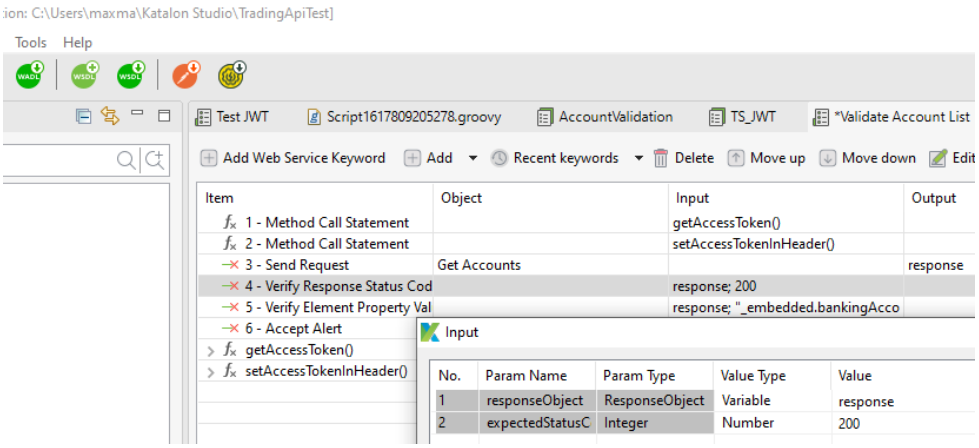


Abbildung 2: Built-In Keywords

Reporting Funktion In Katalon gibt es sogenannte Testsuites. Eine Testsuite ist eine Sammlung von mehreren Testfällen. Wenn eine Test Suite in Katalon gestartet wird, erstellt das Programm automatisch einen Report, welcher auf die Online-Plattform Katalon Analytics hochgeladen wird und dort genauer betrachtet werden kann. In Katalon Analytics sind so alle jemals durchgeführt Testfälle vorhanden und immer aktuell. Damit unterstützt Katalon das Team bei Entscheidungen mit Echzeitdaten. In dem UI können viele verschiedene Statisitken zu den Testfällen angezeigt werden und es eignet sich auch um vergangene Testfälle mit neuen zu vergleichen.

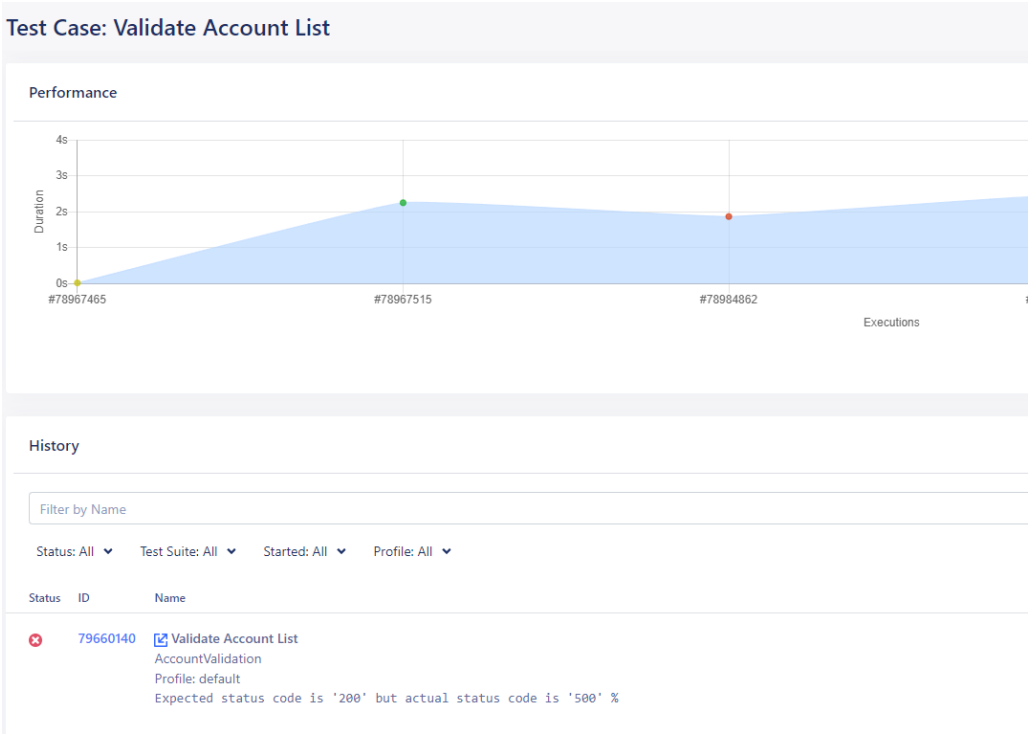


Abbildung 3: Katalon Analytics Ansicht Testfall

Neben dem Upload wird der Report auch noch als HTML, CSV, EXCEL und JSON Datei angeboten. Mit Kommentaren, welche beim Schreiben des Tests festgelegt werden können, einzelne Testschritte definiert werden. Welche dann im entsprechenden HTML Report angezeigt werden.

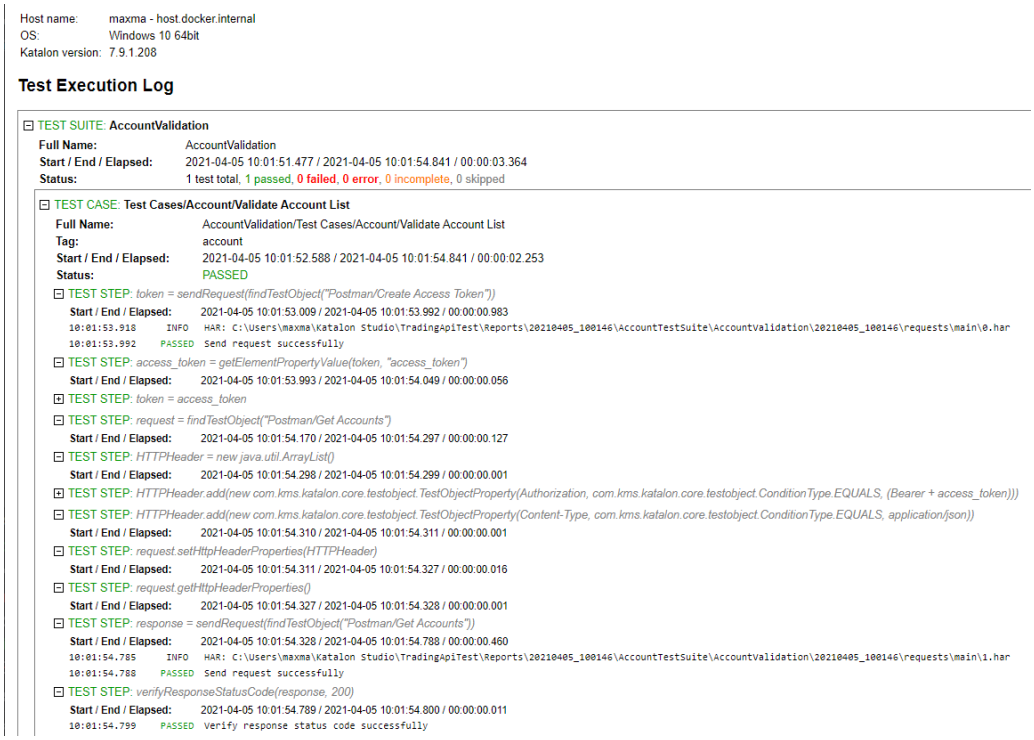


Abbildung 4: Katalon HTML Report

Ergebnis Reporting Funktion

Die Reporting Funktion von Katalon ist sehr positiv zu bewerten. Sobald eine Testsuite beendet wird, werden die Reports lokal erstellt und in die Katalon Analytics Plattform hochgeladen. Dort hat man viele Möglichkeiten, um die verschiedenen Testausführungen miteinander zu vergleichen oder einzeln zu analysieren.

Einbau in CI/CD Pipeline

Der Einbau in die die Gitlab CI/CD war bei Katalon mit Problemen verbunden. Es gibt auf ihrer Homepage nur eine kleine Einleitung (vgl. [12]), welche auf Windows beschränkt ist. Eine Lösung mit Docker wird nicht beschrieben. Dies ist ein grosser Nachteil, da man so auf die Plattform Windows eingeschränkt ist. Mittels eines Bash-Befehls wird die Katalon Runtime im Konsolen Modus gestartet und ausgeführt. Dem Befehl können Parameter mitgebeben werden, um Iterationen durchzuführen oder einen Proxy dazwischen zu schalten.

Bewertung

Die Untersuchung von Katalon hinterlässt einen gemischten Eindruck. as Reporting von Testfällen ist ausgezeichnet gelöst und ansprechend. Es werden automatisch Reports für ausgeführte Test-Cases erstellt. Diese können als HTML, EXCEL oder CSV oder JSON exportiert werden. Ein weiterer Vorteil ist, dass neben API-Tests, Katalon auch für das Testen von User Interfaces und Mobile Apps verwendet werden kann. Beim Erstellen eines Tests für einen geschützten Endpoint und der Einbindung in die CI/CD haben sich viele Nachteile gezeigt:

Funktionalität und Aufwand : gut (+)

Katalon bietet sehr viele Funktionen an, die den Entwickler beim Schreiben von Tests unterstützen. Reports werden automatisch für gelaufene Tests erstellt, ohne dass der Entwickler dafür etwas tun muss. Die Reports können manuell erweitert oder angepasst werden. Die Verwendung von Keywords beschleunigt die Erstellung eines Tests. Leider ist die Authentifizierungsfunktion für unseren Zweck unbrauchbar. Die Hinterlegte Authentifizierung wird im GUI Mode nicht gespeichert. Nach Beenden des Programms musste die Credentials erneut an das Programm übergeben werden. Die Funktion ist so nur für manuelles Testen zu gebrauchen. Die CI/CD Unterstützung für Gitlab ist mangelhaft, da sie nur mit einem Windows-Maschine umgesetzt werden kann. Die Umsetzung der Testfälle hat bei Katalon mehr Zeit in Anspruch genommen als geplant. Mit Hilfestellungen aus dem Web konnten und einem grösseren Aufwand konnten die Testfälle schlussendlich umgesetzt werden.

Dokumentation und Community-Support: schlecht (-)

Dokumentation von Katalon für API Testing ist nicht ausreichend, wenn man einen geschützten Endpoint automatisiert testen will. Im offizielen Forum wurden viele offene Fragen entdeckt die schon lange Zeit nicht beantwortet wurden. Es wird geworben das Tests mit BDD Cucumber und Rest Assured geschrieben werden können, aber es gibt keine Dokumentationen oder sonstige Hilfestellung darüber. Auch eigene Versuche sind gescheitert und es war mir nicht möglich diese Features auszutesten

Wiederverwendbarkeit: gut (+)

Mit sogenannten Workflow Test-Cases können erstellte Tests durch andere Tests aufgerufen werden. Dadurch kann ein Test mehrere andere Tests verwenden. Man hat dadurch die Möglichkeit einen Test-Case wie eine Prozedur zu verwenden. Dies bietet den Vorteil das beispielweise die Authentifizierung in einen Test-Case ausgelagert wird und so von jedem anderen Test verwendet werden kann, ohne den Code duplizieren zu müssen. Im Gegensatz zu Prozeduren können aber keine Parameter übergeben werden und es muss stattdessen mit globalen Variablen gearbeitet werden.

Usability: gut (+)

Die Benutzeroberfläche ist gut durchgedacht und man findet sich schnell zurecht. Die Verwendung der Keywords ist nicht optimal gelöst und konnte erst mithilfe einer Dokumentation korrekt verwendet werden. Die Strukturierung und Gruppierung der Tests ist hervorragend wie auch das eigene UI zur Verwaltung der Reports

Benötigte Skills: schlecht (-)

Es sind Groovy-Script Kenntnisse notwendig, um Tests für eine geschützte API zu schreiben. Der Praxispartner besitzt aktuell nur sehr wenig Mitarbeiter mit Groovy Kenntnissen, da die Sprache bei anderen Systemen kaum zum Einsatz kommt.

1.6.1.2 Postman

Testen eines geschützten Endpoints

Bei Postman konnte ein geschützer Endpoint ohne Probleme und Coding-Skills getestet werden. Postman bietet wie auch Katalon ein UI für die Autorisierung eines Endpoints an. Das UI ist viel eleganter und ausgereifter als bei Katalon und die Informationen können auch dauerhaft gespeichert werden. Der Header wird bei Postman automatisch zum Request hinzugefügt und es war einfach einen Test für einen geschützen Endpoints umzusetzen.

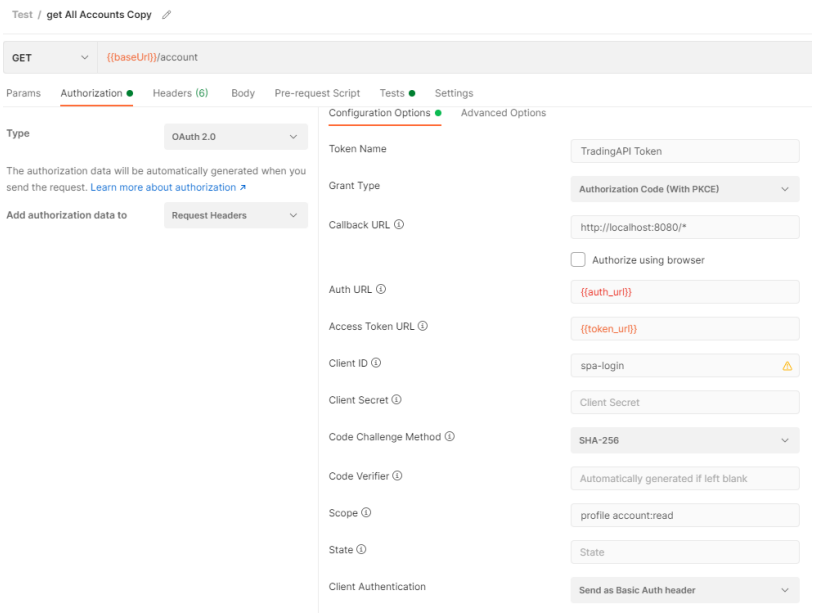


Abbildung 5: Postman GUI für Autorisierung

Für das Schreiben von Tests bietet Postman keine visuelle Benutzeroberfläche an. Es existiert im Gegensatz zu Katalon nur eine Editor-Ansicht. Die verwendete Sprache ist JavaScript. Das Schreiben eines Tests war jedoch wesentlich einfacher als bei Katalon, weil die Test-Syntax von Postman sehr einfach aufgebaut ist. Auch ohne grosse JavaScript Kenntnisse ist es möglich einen Test zu schreiben, da viele Code-Snippets zur Verfügung stehen und so automatisch Code für den Benutzer generieren.

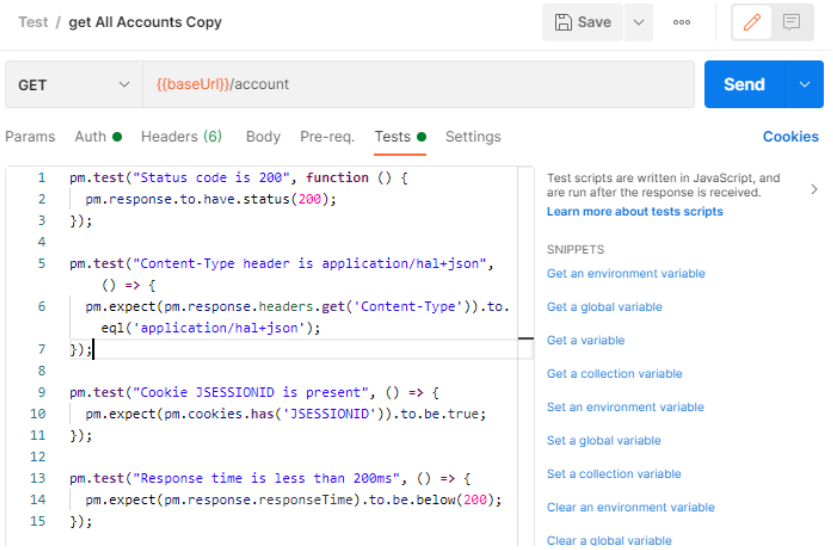


Abbildung 6: Postman Editor um Tests zu Schreiben

Reporting Funktionen

Die Erstellung eines Reports mit Postman mit Postman ist nur im JSON und CVS Format möglich. In Postman gibt es Collections, welche eine Gruppe von Testfällen beinhalten. Für alle Tests in einer Collection kann ein Report erstellt werden, wo ersichtlich ist welche Testfälle erfolgreich waren und welche nicht. In Verbindung mit dem Automatisierungstool „Newman“, welches für den Einsatz in der CI/CD nötig ist, kann aber auch ein HTML Report erstellt werden. Der HTML Report ist sehr ausführlich aufgebaut und man sieht im Gegensatz zum Standard Report von Katalon auch die Header und Bodys der Anfragen und Antworten. Die Möglichkeit einer Erweiterung oder Anpassung des Reports fehlt hingegen.

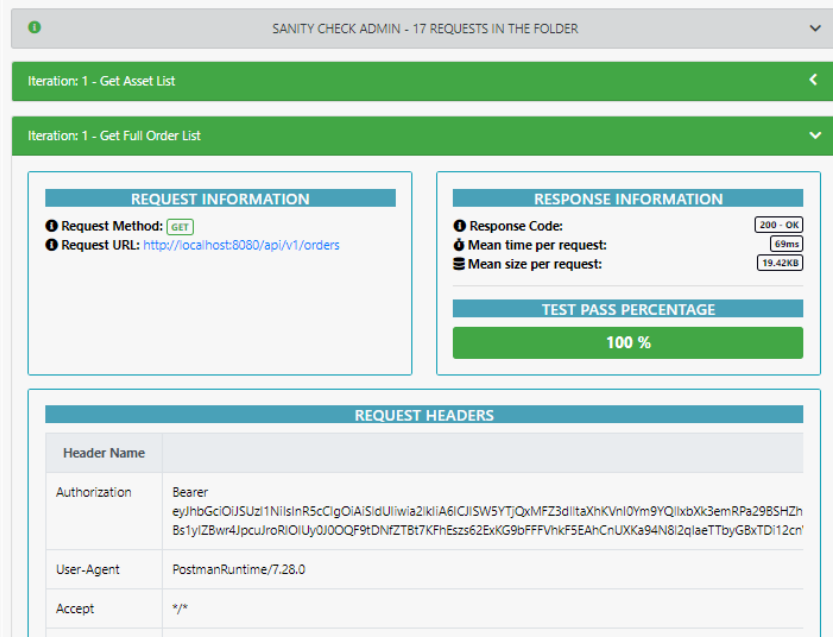


Abbildung 7: Newman HTML Report

Einbau in die CI/CD Pipeline

Der Einbau in die die Gitlab CI/CD ging sehr einfach und schnell. Es gibt im Web zahlreiche Dokumentationen zum Einsatz von Newman in diversen CI/CD Tools. Die Umsetzung mithilfe eines Docker-Images hat ohne Probleme funktioniert. Auch bei Newman kann man die Iterationen mittels Parameter angeben und einen Proxy aktivieren. Das erstellen eines Reports erfolgt ebenfalls durch Parameterangabe. Ein Nachteil von Postman ist das man bei Änderungen der Testfälle immer die gesamte Collection neu als JSON exportieren muss, um diese von Newman ausführen zu lässen.

Bewertung

Ich mochte Postman wegen seiner Einfachheit, der Leichtigkeit der Testentwicklung und die Möglichkeit, die Tests mit dem Newman-Kommandozeilen-Tool zu automatisieren. Es war sehr einfach so die Tests in die CI/CD Pipeline einzubauen. Der grösste Vorteil von Postman gegenüber Katalon ist die Benutzerfreundlichkeit des User-Interfaces.

Ein weiterer Vorteil gegenüber Katalon ist die riesige Community von Postman. Im Web finden sich viel mehr Dokumentationen zu Problemen und Hilfestellung. So war es beispielsweise viel einfacher auch für Postman die Authentifizierung in Form eines Scripts durchzuführen. Viele Entwicklern ist Postman bereits bekannt und es kann auch für manuelle explorative Tests verwendet werden. Der grösste Nachteil von Postman ist die Wiederverwendbarkeit von Scripts. Die geschrieben Scripts können nicht weiter

verwendet werden. Dadurch ist man gezwungen für jeden Test ein neues Test Script zu schreiben und so den Code zu duplizieren. Bei grossen APIs könnte dies problematisch sein, da dadurch auch die Wartbarkeit schwerer wird.

Funktionalität und Aufwand: gut (++)

Die Umsetzung der Testfälle mit Postman war sehr einfach. Postman bietet eine ausserordentliche Möglichkeit an, um sich für eine API zu autorisieren. Das Erstellen eines HTML Reports mit Newman war einfach und auch der Einbau in die CI/CD erfolgte ohne weitere Probleme. Die Reports können aber nicht manuell erweitert oder angepasst werden. Der Aufwand für die Umsetzung der Testfälle mit Postman war geringer als bei Katalon. Es musste sehr wenig nachgeschlagen werden, um die Testfälle umzusetzen. Man ist schnell mit dem Tool vertraut und kann so effektiv arbeiten.

Dokumentation und Community-Support: sehr gut (++)

Die Dokumentationen zu Postman sind sehr ausführlich und man findet im Web viele Beiträge zu Problemstellungen, die auch zahlreich beantwortet wurden.

Usability: sehr gut (++)

Die Benutzeroberfläche von Postman ist ausserordentlich. Das Schreiben eines Tests ist sehr intuitiv gestaltet. Besonders gut gelöst ist auch die Anzeige von Anfragen und Responses in der Postman Konsole, dadurch können Fehler sehr einfach analysiert werden.

Wiederverwendbarkeit: schlecht (-)

Bei Postman ist die Wiederverwendbarkeit von Code stark eingeschränkt, da keine modularen Scripts erstellt werden können, welche von einem Testfall verwendet werden können. Ein Testfall kann keinen anderen Testfall aufrufen wie bei Katalon. Ein Skript für den Bezug eines Access Tokens kann also nicht global gespeichert werden. Es gibt aber die Möglichkeit ein Skript für eine Sammlung für Testfälle zu definieren. Das Skript wird jedoch zwingend von jedem Test in der Sammlung (Collection) aufgerufen. Zum Testen einer API wird normalerweise immer eine neue Collection verwendet, dies bedeutet, dass der Code für die Authentifizierung für jede Collection dupliziert und modifiziert werden muss. Dadurch leidet die Wartbarkeit stark, da bei einer Änderung jede Collection einzeln angepasst werden muss.

Benötigte Skills: gut (+)

Es sind Java-Script Kenntnisse notwendig, um Tests zu schreiben. Der Praxispartner besitzt einige Mitarbeiter mit diesen Kenntnissen, da die Sprache auch bei anderen Applikationen angewendet wird.

1.6.1.3 Rest Assured

Testen eines geschützten Endpoints

Im Gegensatz zu den beiden anderen beiden fixfertigen Lösungen ist bei Rest Assured kein GUI vorhanden. Man benötigt Java Programmierkenntnisse um einen Test zu schreiben. Um die Tests ausführen zu können muss man sein bestimmtes Testing-Framework wie Junit oder Testng verwenden. Das Testen eines geschützten Endpoints ging mit Rest-Assured problemlos und einfach, da sich viele Dokumentationen und Hiilfestellungen im Web finden. Es muss in diesem Fall zuerst ein Request erstellt werden, um ein Access Token zu erhalten um sich am geschützten Endpoint zu authorisieren. Das Token muss dann zwischengespeichert werden und bei den Requests in den Header eingefügt werden. Rest Assured verfügt über Methoden zum Abrufen von Daten aus fast jedem Teil der Anfrage und kann so komplexe JSON-Strukturen auslesen.

```
private String oauth2Payload = "{\n" +
            "  \"client_id\": \"" + userAdminClientId + "\",
            "  \"client_secret\": \"" + userAdminClientSecre
            "  \"audience\": \"http://localhost:8081/realms/
            "  \"grant_type\": \"client_credentials\",\n" +
            "  \"scope\": \"create:account\" \n}";

public String getAccessToken(String payload) {
        return given()
                .contentType(ContentType.JSON)
                .body(payload)
                .post("/token")
                .then().extract().response()
                .jsonPath().getString("access_token");
    }
 @Test
    public void createAccount() {
        userAdminConfigSetup();
        response = given(requestSpecification)
                .body(createUserPayload)
                .post("/accounts")
                .then().extract().response();

        Assertions.assertEquals(201, response.statusCode());
    }
```

Abbildung 8: Rest Assured Code für Autorisierung

Reporting Funktionen

Rest-Assured bietet selbst keine Report-Funktionen an und es müssen weitere Frameworks wie beispielsweise Allure verwendet werden. Aufgrund der benötigen Zeit wurde dies nicht umgesetzt. Die Reporting-Funktion kann aber individuell mit einem Framework umgesetzt werden. Man ist dadurch weniger eingeschränkt als bei fixfertigen Lösungen, hat aber gleichzeitig auch mehr Aufwand, um die Funktionalität anbieten zu können.

Einbau in die CI / CD

Um die erstellten Tests des Java Projekts in der Gitlab CI/CD Pipeline ausführen zu können muss das Projekt zuerst in Gitlab veröffentlicht werden. Die erstellten Tests können mit den üblichen Befehlen zum Ausführen von Unittests in Java ausgeführt werden. Der Befehl varriert je nachdem welches (Junit, TestNg, etc) im Java Projekt verwendet wird.

Erkenntnisse

Mit Rest-Assured sind praktisch keine Limitierungen vorhanden und man kann mit entsprechenden Aufwand damit sein eigenes Testing-Framework erstellen. Bei Rest-Assured hat man im Gegensatz zu Postman und Katalon eine viel besser Wiederverwendbarkeit von Code. Rest-Assured kann auch für Integrationtests verwendet werden und ist vielen Java-Entwicklern bekannt.

Bewertung

Mit Rest-Assured sind praktisch keine Limitierungen vorhanden und man kann mit entsprechenden Aufwand damit sein eigenes Testing-Framework erstellen. Bei Rest-Assured hat man im Gegensatz zu Postman und Katalon eine viel besser Wiederverwendbarkeit von Code. Rest-Assured kann auch für Integrationtests verwendet werden und ist vielen Java-Entwicklern bekannt.

Funktionalität und Aufwand: mässig (o)

Der Aufwand für die Umsetzung der Testfälle ist bei Rest Assured am höchsten, da man für das Reporting und das Aufsetzten des Java-Projekts mehr Zeit benötigt. Rest Assured verfügt über Methoden zum Abrufen von Daten aus fast jedem Teil der Anfrage und Antwort, egal wie komplex die JSON-Strukturen sind. Dies macht das Schreiben von Tests sehr einfach und intuitiv. Wenn der Code nicht sauber strukturiert wird und ausser Kontrolle gerät, kann es zu einer massiven Menge an aufgeblähtem und unnötigem Code führen. Wenn das passiert, steigt der Overhead und damit auch die Kosten.

Dokumentation und Community-Support: gut (+)

Die Dokumentationen zu Rest-Assured sind sehr ausführlich und es gibt viele Foren im Web mit Lösungsvorschlägen zu Problemstellungen.

Usability: mässig (o)

Mit Rest Assured steht dem Test keine Benutzeroberfläche für das Schreiben von Tests zur Verfügung. Das Testen und Validieren von REST-Diensten in Java ist schwieriger als in dynamischen Sprachen wie Javascript und Groovy. Die Rest Assured Syntax vereinfacht das Schreiben von Tests in Java.

Wiederverwendbarkeit: sehr gut (++)

Da Rest Assured eine Java-Bibliothek ist hat man alle Vorteile einer objektorientieren Sprache. Es können modulare Klassen und Konzepte wie Vererbung verwendet werden, die zusammen eine viel höhere Wiederverwendbarkeit bieten, als beispielweise die Wiederverwendung eines Test-Cases. Ausserdem ist die Wartung des Codes bei einer objektorientieren Sprache einfacher als bei skriptbasierten Sprachen wie JavaScript.

Mit einer guten Code Strukturierung und modularen Klassen kann damit eine höhere Wiederverwendbarkeit erreicht werden.

Benötigte Skills: gut (+)

Java wird auch bei der Entwicklung der API verwendet und ist beim Praxispartner weit verbreitet. Dies hat den Vorteil, dass sich qualifizierte Mitarbeiter bereits im Unternehmen

befinden. Da jedoch keine Benutzeroberfläche wie bei den anderen beiden Tools vorhanden ist benötigt der Entwickler tiefere Kenntnisse.

1.6.1.4 Entscheid

Bewertung

Aus den gewonnen Erkenntnissen zu den einzelnen Tools geht hevor, dass alle Tools die Anforderungen erfüllen. Die Analyse von Rest-Assured hat gezeigt, dass ein individuelles Framework alle Aufgaben ebenfalls erfüllen kann. Jedoch ist der Aufwand viel grösser u die gleiche Funktionalität wie die fixfertigen Lösungen anzubieten. Da für API-Tests ca 20% des gesamten Testaufwands investiert werden soll, eignen sich.Rest Assured deshalb nur wenn bereits ausgereifte Kenntnisse vorhanden sind.. Die Wiederverwendbarkeit ist bei Rest-Assured jea Mustandoch am Grössten.

Katalon hat eine sehr gute Reporting-Funktion konnte aber beim Schreiben von Test und der Automatisierung nicht überzeugen. Bei Katalon war der Lernaufwand höher und die Umsetzung eines Tests viel schwieriger als bei Postman. Beide Tools verwenden Konzepte wie Globale Variablen und Test-Collections bzw Test-Suites. Das User-Interface von Postman ist extrem benutzerfreundlich, weshalb auch viel schneller Fortschritte erzielt werden konnten. Ein ausschlaggebender Punkt war ausserdem die Dokumentation und der Support. Bei Katalon sind nur Dokumentation für triviale Testfälle vorhanden. Dokumentation zum Scripting oder Beispiele für die Integration von BDD findet man nicht. Bei Postman hingegen konnte zu jedem gesuchten Thema eine Dokumentation oder Hilfestellung gefunden werden

	Katalon	**Postman**	**Rest Assured**
Funktionalität und Aufwand	Gut (+)	Gut (+)	Mässig (o)
Dokumentation und Community-Support	Schlecht (-)	Sehr gut (++)	Gut (+)
Usability	Gut (+)	Sehr gut (++)	Mässig (o)
Wiederverwendbarkeit	Gut (+)	Schlecht (-)	Sehr gut (++)
Benötigte Skills	Schlecht (-)	Gut (+)	Gut (+)
Platzierung	**3**	**1**	**2**

Die Bewertung der einzelnen Kandidaten zeigt das Postman bei den festgelegten Kritieren am meisten überzeugen konnte. Es wurde deshalb mit dem Praxispartner zusammen entschieden das Tool Postman für das Schreiben der End-to-End Tests zu verwenden.

1.6.2 Szenario der API klären

Vorhaben / Ziel

Die Daten und Operationen, welche durch die verschiedenen Endpoints einer API transponiert werden, spielen eine zentrale Rolle für die Angriffsfläche einer API. Damit die Beispiel-API ein möglichst reales Szenario abbildet musste mit dem Praxispartner geklärt werden, welches Szenario die API abbilden soll und was ihre Datenbasis ist.

Szenario

Es wurde zusammen mit dem Praxispartner festgelegt, dass die Prototyp-API das Geschäftsmodell Trading simulieren soll. Bei der API handelt es um sich um eine Partner-API. Partner-APIs sind APIs, die für bestimmte Geschäftspartner offengelegt werden. Über einen API-Gateway sollen bestimmte Benutzer Zugriff erhalten. Im Gegensatz zu völlig offenen APIs, gibt es für die Benutzung ein Validierungsverfahren, um den Zugang zur API zu erhalten. Die API simuliert das Suchen, Kaufen und Verkaufen von Wertpapieren. Im weiteren Gespräch mit dem Praxispartner hat sich die Frage nach der Datenbasis gestellt. Dabei haben sich die nachfolgenden zwei Varianten gebildet:

Variante 1: Verwendung des Buchungssystems (Keine Datenbank)

Bei dieser Variante wird die API des Buchungssystems von der Beispiel-API verwendet. Bei dieser Variante verbindet sich der Data Access Layer nicht direkt mit einer Datenbank, stattdessen werden die gekapselten API-Calls über die API des Buchungssystem an eine anonymisierte Testinstanz weitergeleitet. Der Vorteil bei dieser Variante ist, dass hierbei aktuell bestehende Schwachstellen der API des Buchungssystem aufgezeigt werden können. Dies ist aber auch gleichzeitig ein Problem, da dies die Publikation der Arbeit erschwert und Verträge zur Geheimhaltung erfordern würde. Die Entwicklung muss ausserdem zwingend auf einem Firmengerät erfolgen und das Deployment ist nur in der Infrastruktur des Praxispartners erfolgen darf.

Variante 2: Eigene Datenbank in abstrahierter Form

Es wird eine eigene Datenbank aufgebaut, welche die notwendigen Daten und Informationen als Testdaten bereitstellt. Durch die stark abstrahierte Form sind Rückschlüsse auf das Buchungssystems des Praxispartners ausgeschlossen. Die Datenbank des Buchungssystem ist äussert komplex, was das Definieren von allgemeinen Security Tests unnötig schwierig macht. Bei einer eigenen Datenbank können ausserdem bewusst Schwachstellen eingebaut werden, um aufzuzeigen wie diese wieder geschlossen werden können.

Entscheid

Aufgrund der beschriebenen Einschränkungen bei der Verwendung der Variante 1, wurde mit dem Praxispartner zusammen entschieden die Variante 2, also eine eigene Datenbank umzusetzen. Das finale Ziel der Arbeit ist es, eine API auf ihre Sicherheit zu testen. Der zusätzliche Aufwand, der bei der Variante 1 entsteht, übersteigt den daraus folgenden Nutzen, da die Datenbasis für die Sicherheitstests eine untergeordnete Rolle spielt. Beim Testen der API ist die zentrale Frage, wie die API das entsprechende Datenmodell abstrahiert und welche Methoden und Zugangspunkte durch sie bereitgestellt werden.

1.6.3 API Security Grundlagen

Inhalt

In diesem Kapitel werden einige Grundlagen zur API-Sicherheit vorgestellt.

Die Grundlagen werden zum Verständnis der Thematik dieser Arbeit und insbesondere für die im Verlauf dieser Arbeit vorgestellten Bedrohungsmodellierung benötigt.

Das Kapitel beinhaltet die nachfolgenden drei Unterkapitel:

- Traditionelle vs moderne Webapplikationen
- Auswirkungen auf die Sicherheit
- Angriffe auf APIs
- API Security Testing

Einleitung

Das Open Web Application Security Projekt (OWASP) ist eine gemeinnützige Online-Community, welche Jahr für Jahr Dokumentationen, Methoden, Tools und Technologien zur Sicherheit von Webapplikationen publiziert. Seit vielen Jahren ist die OWASP Top 10 List die massgebliche Liste mit Informationen zu Schwachstellen in Webapplikationen. Die Angriffsszenarien im Bedrohungsmodell für eine traditionelle Webapplikation unterscheiden sich von denjenigen einer API. Es ist ein Irrtum zu glauben, man könne APIs mit denselben Methoden sichern kann, wie herkömmliche Webapplikationen. APIs unterscheiden sich grundlegend von Webseiten und haben ein ihr eigenes Risikoprofil. Aus diesem Grund publiziert OWASP seit 2019 eine eigene Top 10 Version für Sicherheitsschwachstellen speziell für APIs.

1.6.3.1 Traditionelle und moderne Webapplikationen

Einleitung

Gemäss OWASP haben sich in den letzten Jahren die Entwicklung von Applikation signifikant verändert. Es wird zwischen Traditionellen Webapplikation und Modernen Applikationen unterschieden. Um die neuen Sicherheitsanforderungen an eine API zu verstehen, muss man verstehen wie sich die Entwicklung von Applikationen in den letzten Jahren verändert haben.

Microservices vs. Monolith

Bei traditionellen Webapplikationen sendet der Client einen HTTP Anfrage an das Backend, um eine komplette HTML Seite zu erhalten. Der Webserver (Backend) wird die Anfrage bearbeiten und Daten von einem Datenbank-Server auslesen. Anschliessend wird die HTML Seite lokal auf dem Webserver erstellt und dann an den Client Browser gesendet. Der Webserver verwaltete den Status des Benutzers mittels Cookies. Viele Angriffe waren bei traditionellen Applikationen auf diese Cookies ausgelegt.

Unternehmen haben sich lange Zeit auf monolithische Anwendungen verlassen, um ihren Betrieb zu führen und ihren Kunden verschiedene Services anzubieten. Monolithische Applikationen sind in sich geschlossen. Die verschiedenen Komponenten der Applikation sind stark miteinander verbunden und somit abhängig voneinander. Bei einer Änderung in dieser eng gekoppelten Architektur muss jede einzelne Komponente bereit stehen damit der Code kompiliert oder ausgeführt werden kann. Bei einem Release wurde immer die ganze Applikation neu ausgeliefert, auch wenn nur eine bestimmte Komponente

angepasst wurde. Dies hatte zur Folge das Releases für die Applikation nur alle paar Monate erfolgten.

Moderne Applikationen setzten sich aus vielen einzelnen Microservices zusammen. Die Microservices-Architektur ist ein Architekturstyle, bei dem eine Applikation in viele separate, aber miteinander verbundene Komponenten, sogenannte Microservices, aufgeteilt wird. Jede Microservice hat seine eigene Logik und führt einen eigenen Prozess aus, der mit den anderen Komponenten über eine API kommuniziert. [13]

APIs stellen damit Microservices für den Verbraucher zur Verfügung, indem sie die Regeln, Befehle und Protokolle für die Kommunikation definieren. Es ist deshalb wichtig zu verstehen wie Microservices funktionieren.

Heutzutage setzten Unternehmen vermehrt auf modulare Komponenten, die einfach ausgetauscht werden können. Ein einzelnes Modul stellt einen Microservice dar. Er kann verändert werden, ohne dass andere Microservices davon betroffen sind.

Microservice bieten eine spezialisierte feinkörnige Zusammenarbeit, die das Architekturmodell ausmacht. Innerhalb der Applikation übernimmt ein Microservice eine bestimmte Aufgabe. Beispielsweise die Authentifizierung von Benutzern, die Generierung eines bestimmten Datenmodells oder die Erstellung eines bestimmten Berichts. Microservices sind oft mit verschiedenen Frameworks in unterschiedlichen Sprachen entwickelt. Microservices passen in jede Art von Applikation und kommunizieren über APIs miteinander, um ein gemeinsames Ziel zu erreichen. [14]

Microservices sind aus den oben genannten Gründen ein beliebter Weg, um Webanwendungen zu erstellen. Zusammenfassend lässt sich sagen, dass eine Microservice-Architektur ab einer bestimmten Applikationsgrösse es einfacher, zuverlässiger und schneller macht, einzelne Teile einer Applikation und damit die Applikation selbst zu erstellen und zu bearbeiten.

Im Gegensatz zu den alten Monolithen werden Microservices nicht mehr in auf selbstverwaltete Server ausgeliefert. Stattdessen werden Container Plattformen wie beispielsweise Docker oder Kubernetes verwendet. Diese beschleunigen das Ausliefern von neuen Funktionalitäten und damit auch den Entwicklungsprozess, da man sich nicht mit der Konfiguration und Verwaltung von Servern herumschlagen muss.

Microservice-Architektur Fallstudie

In einem Artikel von Dzone (vgl. [15]) werden die Änderungen durch eine Microservice-Architektur anhand einer Fallstudie nochmals einfach veranschaulicht:

„Wie viele Startups begann auch Uber mit einer monolithischen Architektur, die für ein einziges Angebot in einer einzigen Stadt gebaut wurde. Eine einzige Codebasis zu haben, schien zu dieser Zeit sinnvoll und löste die Probleme von Uber. Als Uber sich jedoch weltweit zu expandieren begann, sind viele neue Probleme im Bereich der Skalierbarkeit und der Integration aufgetaucht. Alle Funktionen mussten immer wieder neu erstellt, bereitgestellt und getestet werden, um eine einzige Funktion zu aktualisieren. Die Behebung von Fehlern wurde in einem einzigen Repository schwierig, da die Entwickler den Code immer wieder anpassen mussten. Weiter war die Skalierung der Funktionen bei

gleichzeitiger Einführung neuer Funktionen weltweit nur schwer gemeinsam zu bewältigen.“ [15]

Abbildung 9: Monolithische Architektur von Uber [15]

Um die oben genannten Probleme zu lösen hat sich Uber dazu entschieden seine Architektur zu verändern. Die monolithische Architektur wurde in mehrere Services aufgebrochen, um daraus eine Microservice-Architektur zu bilden

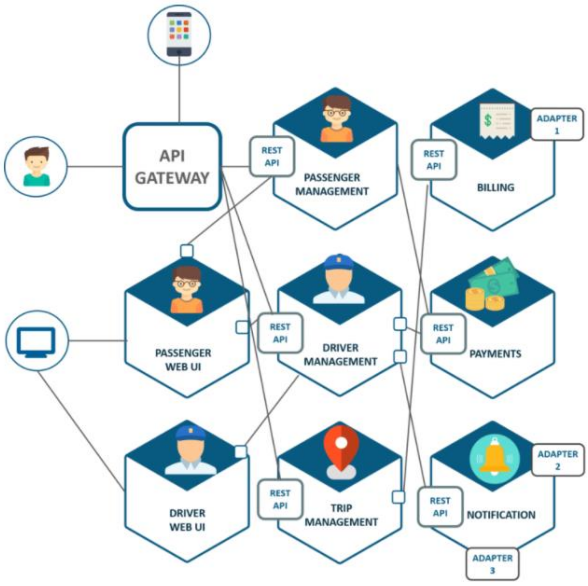


Abbildung 10: Microservice Architektur von Uber [15]

Die wichtigste Änderung ist, dass nun jeder Microservice selbst über eine API verfügt, über die er erreichbar ist. Die APIs sind nicht mehr direkt erreichbar für den Benutzer. Es wurde ein API Gateway eingeführt, über welches die Benutzer sich verbinden, um mit einem entsprechenden Microservice kommunizieren zu können. Jeder Microservice führt eine bestimmte Aufgabe durch wie beispielsweise die Verwaltung von Benutzern oder das Durchführen einer Zahlung. Die Services sind lose gekoppelt, sodass eine Änderung in einem Microservice keine Anpassung eines anderen Microservices erzwingt. Die Microservices können völlig eigenständig entwickelt und bereitgestellt werden.

Veränderung der Clients

Der Client einer Webapplikation war früher immer ein herkömmlicher Browser, wie beispielsweise Chrome oder Firefox. Er war nur für das Empfangen und Anzeigen von HTML-Seiten zuständig und informierte den Webserver mittel Statusinformationen über den User.

Der Boom von mobilen Apps und Single Page Applikationen haben neue Clients hervorgebracht und die Aufgaben neu verteilt:

- Die Clients, welche sich mit dem Backend über eine API verbinden waren traditionell der Browser. Heutige APIs bedienen eine wesentlich grössere Anzahl von Clients, da neben den traditionellen Browsern auch Mobile-Apps, IOT-Geräte und andere Webapplikation.
- Anstelle eines HTMLs benötigten moderne Single Page Applikationen heutzutage eine JSON-Datei, um Teile der Seite zu aktualisieren. Anstelle eines einzelnen HTTP-Aufrufs zum Laden einer HTML-Seite, werden heute duzende Aufrufe an die API vom Client gesendet, um einzelne JSON Dateien zu erhalten, um so Teile der Seite zu aktualisieren. Der Client benötigt spezifischere Informationen als bei traditionellen Applikationen.
- Der Client besitzt viel mehr Kontext, da sie neu die Session des Users verwalteten und daher genau wissen, welche Teile sie benötigen. Dadurch wird ein Request spezifischer, was automatisch zu mehr Parametern führt. Der Webserver, also die API ist bei modernen Applikationen dafür zuständig Daten für einen spezifische Anfrage von einer Datenquelle auszulesen und sie dem Client bereitzustellen.
- APIs sind heutzutage vielmehr wie ein Proxy zwischen einer Datenbank und dem Client zu verstehen [16]. Das Rendering, also der Prozess zum Erstellen einer visuellen Seite für den Browser hat sich durch die weit verbreiteten Single Page Applikationen vom Backend zum Client verschoben.

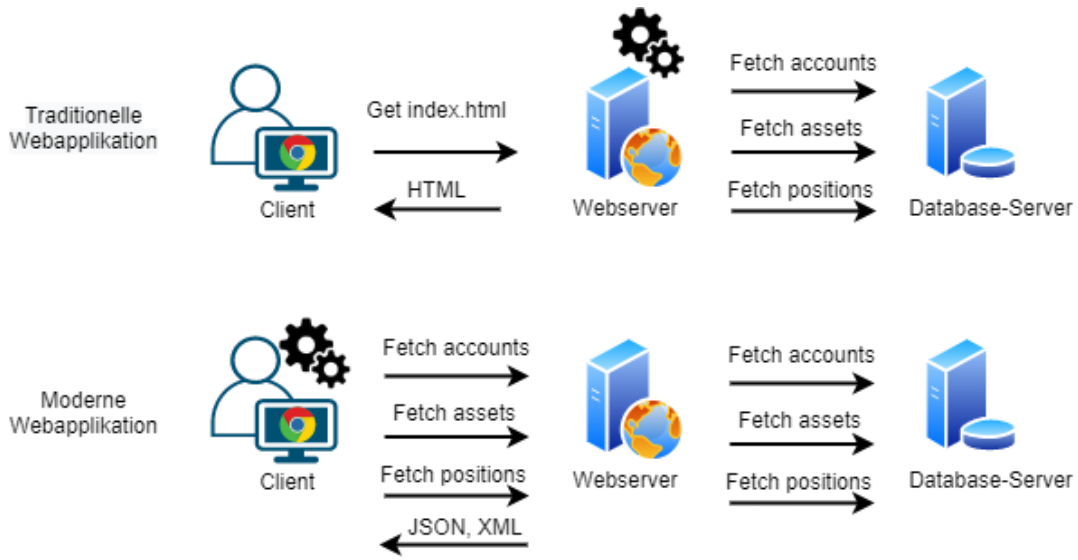


Abbildung 11: Traditionelle vs Moderne Applikation

- Moderne Applikationen bieten eine schwächere Abstraktion als traditionelle Applikationen. Eine API ist eine Abstraktionsschicht, da sie die zugrunde liegende Implementierung und ihre Komplexität verbirgt. Bei einem Blackbox Test und für

einen Angreifer ist es wichtig, dass er das Backend einer Applikation versteht. So ist es für ihn einfacher oder überhaupt möglich einen bestimmten Angriff vorzubereiten und durchzuführen. Bei traditionellen Applikationen war es viel schwieriger für einen Angreifer die Backend-Logik zu analysieren. Eine API gibt einem Angreifer viel mehr Informationen über die dahinterliegende Implementation preis.

- Der Client und die API kommunizieren Objekte im JSON-Format, sie besitzen dadurch eine gemeinsame Sprache. Häufig ist das JSON-Objekt im Backend dasselbe wie beim Client. Durch die öffentlich zugänglichen Dokumentationen benötigt ein Angreifer unter Umständen nicht einmal Zugriff, um die dahinterliegende Applikation zu verstehen.

RESTful API

Im Fokus dieser Arbeit steht die Entwicklung und das Testing einer Restful API. Es handelt sich dabei um eine API, welche die Prinzipien von REST erfüllt.

Eine API an sich nichts Neues. Bereits 1998 hat Microsoft die XML-RPC [17] Spezifikation veröffentlicht, welches HTTP für den Transport und XML als Protokoll verwendet. Mit der Einführung neuer Funktionalitäten entwickelte sich der Standard zu dem, was heute SOAP ist.

SOAP und REST sind zwei API-Stile, welche die Datenübertragung aus unterschiedlichem Blickwinkel betrachten. Simple Object Access Protocol (SOAP) ist ein offizielles Protokoll ist, verfügt es über strenge Regeln und erweiterte Sicherheitsfunktionen. es erfordert mehr Bandbreite und Ressourcen, was zu langsameren Seitenladezeiten führen kann. [18]

Representational State Transfer (REST) ist ein Architekturstil und wurde entwickelt, um die Probleme von SOAP zu lösen. [19]. REST soll die wünschenswerten Eigenschaften, wie Leistung, Skalierbarkeit und Zuverlässigkeit der Microservice-Kommunikation verbessern REST ist im Gegensatz zu SOAP kein Protokoll Es besteht nur aus losen Richtlinien und lässt Entwicklern die Möglichkeit, die Empfehlungen auf ihre eigene Weise zu implementieren. REST erlaubt im Gegensatz zu SOAP mehrere verschiedene Dateiformate (HTML, JSON, XML, Python, PHP, Plain Text). JSON ist das meistverwendete Format da es sprachunabhängig ist und für den Menschen lesbar ist. REST ist eine leichtgewichtige Architektur und hat deshalb die bessere Performance als das SOAP Protokoll. Aus diesem Grund ist es in der heutigen Zeit, wo jede Sekunde zählt, so unglaublich populär geworden. [18].

Damit eine API sich REST-API oder RESTful API nennen darf, müssen die sechs Prinzipien (vgl. [19]) des Architekturstil REST erfüllt sein:

- Client-Server-Architektur
- Zustandslosigkeit
- Cachefähigkeit
- Mehrschichtiges System
- Code auf Anfrage
- Einheitliches Interface

1.6.3.2 Auswirkungen auf die Sicherheit

Vorhaben / Ziel

Dieses Kapitel thematisiert die Auswirkung auf die Sicherheit einer API, welche durch die thematisierten Veränderungen aus dem vorherigen Kapitel entstehen.

Einleitung

Wie bereits in der Einleitung erwähnt haben machen APIs heute über 80% des Web-Verkehrs aus. Man kann daraus folgen, dass die Sicherheit von Webapplikation neu die Sicherheit von APIs ist. In einer Studie aus dem Jahr 2019 schreibt Gartner [3], dass bereits 40 % der Angriffe auf Webapplikationen über exponierte APIs erfolgen. Ausserdem prognostizierte Gartner, dass diese Zahl bis Ende 2021 auf 90 % steigen wird.

API Security

Ivan Novikov, der CEO von Wallarm, hat in einem Bericht (vgl [20]) des Forbes Magazin API Security folgendermassen definiert:

Am einfachsten stellt man sich die Angriffsfläche einer API vor, indem man sie in Schichten aufteilt:

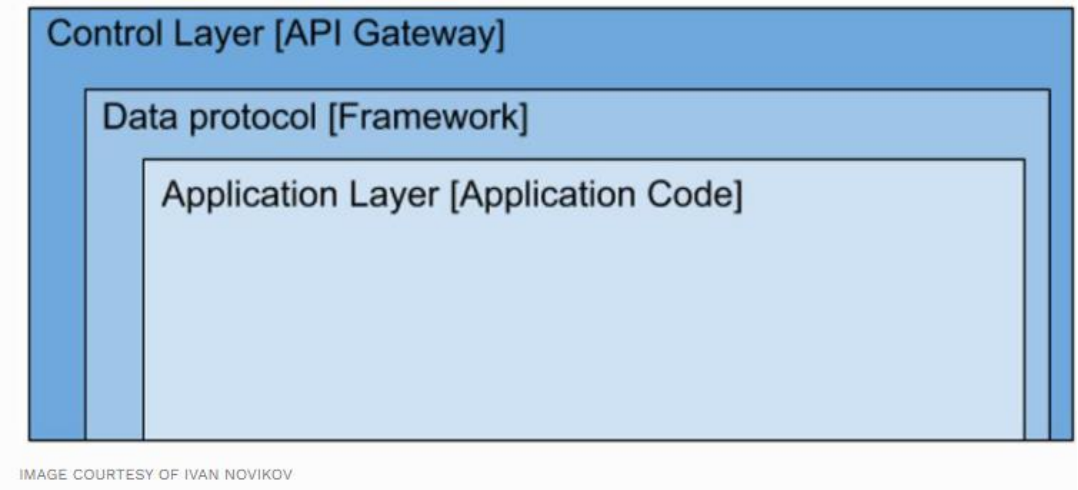


Abbildung 12: Angriffsfläche API [20]

Der Control Layer bezieht sich auf die Autorisierung und grundlegende Kontrollen wie Anfrageraten. Die Autorisierung wird meistens über eine API Gateway implementiert, kann aber auch auf API-Ebene erfolgen.

Der Data Protocol Layer liegt zwischen den beiden oben genannten. Alle API-Anfrageformate, wie XML, JSON oder gRPC, sind Formate zur Datenserialisierung. Sie beschreiben, wie ein Objekt in ein String Format gebracht werden kann. Angriffe auf XML sind wegen der XXE-Schwachstelle (XML External Entity) bekannt. Während JSON

wegen der Jackson-Databind-Remote-Code-Execution-Schwachstelle für Dot Net-Anwendungen bekannt.

Der Application Layer ist mit traditionellen Web-Anfragen gleichzusetzten. Es gibt nur zwei zusätzliche Schichten für die API und dahinter kommt eine gewöhnliche Web-Anfrage.

Dieses Modell zeigt auf, dass die üblichen Schwachstellen der OWASP Top 10 für Webapplikationen nach wie vor vorhanden sind. Der grosse Unterschied besteht darin, dass die klassischen Web-Request unter der Haube eines Dateiformats wie JSON oder XML liegt. Für einen Angreifer ergeben sich dadurch neue Angriffsmöglichkeiten zur Umgehung von Sicherheitslösungen. Es wurde gezeigt das API-Sicherheit durch drei Schichten dargestellt werden kann, wobei der Application Layer der Sicherheit von klassischen Webapplikationen entspricht. [20].

Erweiterte Angriffsfläche

Clients werden immer mächtiger und erhalten durch APIs einen viel spezifischeren Zugriff auf das Backend als bei traditionellen Webapplikationen. Damit hat ein Angreifer eine viel grössere Angriffsfläche. Jeder Endpoints und jeder Parameter, den eine API anbietet, ist eine potenzielle Schwachstelle und erhöht damit das Risiko eines Angriffs.

Die einzelnen Microservices haben jeweils ihre eigene RESTful API, über welche sie dem Internet ausgesetzt sind. Durch die neuen Endpoints legt eine Organisation einen grösseren Teil ihrer Geschäftslogik frei, welche bis anhin verbogen war

Eine unzureichende Kontrolle der Versionierung und Bereitstellung führt dazu, dass veraltete API-Versionen neben aktuellen Versionen laufen. Ältere Versionen sind ein leichteres Ziel und ermöglichen oft den Zugriff auf Daten. Das gibt Angreifer eine wesentlich höhere Angriffsfläche und erhöht das Risiko eines Angriffs.

Neue Clients

Die meisten APIs für Microservices werden von mobilen Anwendungen oder von anderen Diensten einer Softwarekomponente verwendet. Die bisherigen Lösungen zum Erkennen von Botnetzen haben sich stark auf die Analyse von Browser Informationen, sogenanntes Fingerprinting (vgl. [21]). Es ist daher schwieriger geworden, automatisierte Angriffe auf API-Endpunkte zu erkennen.

Client Security ist ein eigenes Thema. Die OWASP Top 10 Schwachstellen für Webapplikationen gelten immer noch für Webapplikationen, die z.B. mit Angular entwickelt werden. Mobile Apps besitzen ebenfalls ihr eigenes Risikoprofil, welches man adressieren muss.

Neue Anforderungen an Authentifizierung und Autorisierung

Bei traditionellen monolithischen Applikationen haben Unternehmen sich darauf konzentriert die äusserste Schicht, welche dem Internet ausgesetzt ist, abzusichern. Dies ist bei modernen Applikationen nicht mehr möglich. Die Microservices müssen untereinander kommunizieren, was die Komplexität der Zugriffskontrolle erhöht. Eine grössten Herausforderung ist deshalb die Implementierung von Authentifizierung und Autorisierung von Microservices. Dies macht die APIs zu einem attraktiven Angriffsvektor für einen Täter, der gestohlene Anmeldedaten validieren oder bereits validierte Konten ausnutzen möchte. Eric Boersma schreibt in seinem Artikel (vgl. [14]) über die Sicherheitsfallen bei der Migration von Monolithen zu Microservices, zur Authentifizierung folgendes:

“Authentication between microservices isn’t something that teams think about when they’re coming from a monolith. You don’t need to authenticate yourself when calling a function in code between libraries on a server. Microservices are a totally different situation. It’s critical to make sure that each service is able to authenticate not just the server requesting data, but also the context of the logged-in user. In a monolith, this is information that’s just stored in memory. On a microservice, that’s information that has to be passed with each request.” [14]

Vorhersehbare Endpoints

Die Anwendung der REST Prinzipien führt zu gut strukturieren APIs, welche die Eigenschaft haben, sich selbst zu beschreiben. Ein Benutzer erkennt so anhand der eindeutigen URL, den Eingabeparametern und der Antwort, intuitiv wie die API zu verwenden ist. Die Transparenz, die eine API über das Backend anbietet, gibt Angreifern oft gefährliche Hinweise, die zu Angriffen führen, welche sonst nicht entdeckt worden wären. Es ist einfacher, administrative Funktionen in APIs zu finden: Anstatt die URL zu erraten ändert ein Angreifer beispielsweise einfach die HTTP-Methode von GET auf DELETE.

Erkenntnisse

Die Unterschiede von traditionellen und mobilen Webapplikationen zeigen auf, weshalb für eine API ein eigenes Risikoprofil erstellt werden muss. Es wurde gezeigt das sich die Angriffsfläche für einen Kriminellen vergrössert hat und die Anforderungen an die Sicherheit gewachsen sind. Die traditionellen Schwachstellen für Webapplikationen gelten noch immer. Jedoch bieten APIs eine neue Angriffsfläche und Schwachstellen für Angreifer.

1.6.3.3 Angriffe auf APIs

Vorhaben / Ziel

Dieses Kapitel setzt sich mit den Angriffen auf APIs auseinander. Zu Beginn wird die Motivation eines Angreifers auf eine API speziell im Hinblick auf eine API im finanziellen Sektor. Im Kontext auf das beschriebene Business Szenario werden die Ziele eines möglichen Angreifers erläutert und verschiedene Angriffsvektoren thematisiert.

Einleitung

Der „State of the Internet“ Sicherheitsbericht 2020 für Unternehmen aus der Finanzbranche von Akamai (vgl. [22]) hat aufgezeigt, dass Kriminelle ihre Angriffsstrategie geändert haben und nun hauptsächlich APIs gezielt angreifen. Der Bericht legt dar, dass bei Organisationen im Finanzdienstleistungssektor bis zu 75% aller Credential Stuffing Angriffe direkt auf APIs erfolgen. Das zeigt eindeutig die Verlagerung der Ziele von Angreifern weg von allgemeinen Anmeldeseiten hin zu API-Anmeldung. Akamai verzeichnete zwischen Dezember 2017 und November 2019 insgesamt 85,4 Milliarden Fälle von Anmeldedatenmissbrauch. Fast 20 % (16,5 Milliarden) davon betrafen Hostnamen, die eindeutig als API-Endpunkte identifiziert wurden. Von diesen zielten 473,5 Millionen Angriffe auf Organisationen in der Finanzdienstleistungsbranche. Zwischen November 2017 und Oktober 2019 waren mehr als 40 % der direkten DDoS-Ziele in der Finanzdienstleistungsbranche angesiedelt.

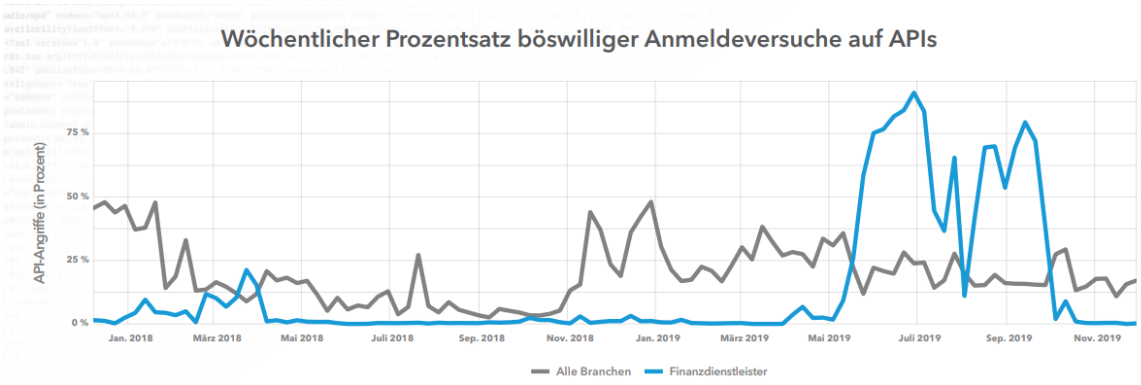


Abbildung 13: böswillige Anmeldeversuche auf APIs [22]

Laut dem am 3.Feb 2021 veröffentlichten Bericht von Salt Security (vgl. [23]) hatten 91 % der Organisationen im vergangenen Jahr Sicherheitsprobleme im Zusammenhang mit APIs. Die häufigsten Probleme waren Schwachstellen in der API mit 54 %, gefolgt von Authentifizierungsproblemen mit 46 %, Bot-Angriffe mit 20 % und Denial-of-Service (DoS) Angriffe mit 19 %. 80% der Unternehmen glauben nicht, dass sie API-Angriffe wirkungsvoll verhindern können. Die Umfrage von Salt Security ergab, dass 66% der Unternehmen den Rollout einer neuen Anwendung in die Produktion aufgrund von Bedenken hinsichtlich der API-Sicherheit nach Hinten verschoben haben. Befragte, die alle über Web Application Firewalls (WAFs) und API-Gateways verfügten, hatten alle mehrere API-Angriffe pro Monat erlebt, was bedeutet, dass die API-Angriffe an diesen Sicherheitstools vorbeigingen. Tatsächlich entgehen WAFs und API-Gateways laut Salt 90 % der OWASP API Security Top 10-Bedrohungen. Noch schockierender ist jedoch, dass 9 % der Befragten zugaben, dass sie API-Angriffe nicht identifizieren können.

Mehr als ein Viertel der Unternehmen betreibt kritische API-basierte Anwendungen ohne Sicherheitsstrategie und weitere 27 % der Unternehmen haben nur eine grundlegende

Strategie für API-Sicherheit. Nicht korrekt geschützte APIs kommen häufiger vor als man annehmen würde. Laut dem Salt Security-Report fehlt 82 % der Organisationen das Vertrauen in die Kenntnis von API-Details, z. B. ob die APIs persönlich identifizierbare Informationen (PII) wie beispielsweise geschützte Gesundheitsinformationen und Daten über den Besitzer einer Karte enthalten, und 22 % der Unternehmen geben an, dass sie keine Möglichkeit haben, zu wissen, welche APIs sensible Daten offenlegen.

Ziele eines Angreifers

Die Ziele eines Angriffs auf eine API können je nach Business Case unterschiedlich sein. Im nachfolgenden werden einige mögliche Ziele eines Angreifers thematisiert:

Verkauf von Daten

Der Kauf und Verkauf von gestohlenen Bankkarten und Anmeldedaten für ist unter Kriminellen weit verbreitet.

Identitätsklau

Ein Angreifer kann sich mit gestohlenen Daten als eine andere Person ausgeben und in ihrem Namen beispielsweise Transaktionen durchführen, um sich finanziell zu bereichern oder Spuren von sich zu verschleiern.

Zugriff auf Ressourcen anderer Benutzer.

Der Angreifer hat sich zum Ziel gesetzt auf Ressourcen von anderen Benutzern der API zuzugreifen und so Informationen über den Benutzer zu erhalten, welche wieder verkauft werden können.

Manipulation von Daten

Der Angreifer hat sich zum Ziel gesetzt die Daten und Informationen einer API gezielt zu manipulieren, um sich beispielsweise finanziell zu bereichern, indem er eine Transaktion entsprechend beeinflusst. Der Angriff zielt auf die Nutzung einer manipulierten Ressource ab und nicht auf die Zerstörung.

Daten sammeln über das Unternehmen, welche die API betreibt.

Viele Angriffe haben das Ziel, Informationen über ein Unternehmen zu sammeln. Bei einer API prüfen Angreifer beispielsweise ob die Endpoints zusätzliche Informationen senden, welche im Frontend nicht angezeigt werden und verstehen so die Backend Applikation besser. Dies können auch vermeintlich unwichtige Informationen wie z.B. die Funktion des Mitarbeiters, interne E-Mail-Adressen oder eine Telefonnummer. Mit diesen Informationen wird dann häufig in einem weiteren Schritt versucht an Anmeldedaten zu gelangen.

Microservice / Server der Unternehmung lahmlegen

Ein Angreifer setzt sich zum Ziel Ressourcen offline zu nehmen oder die korrekte Nutzung der API oder dahinterliegender Services zu verhindern.

Angriffe auf eine API

Ein API-Angriff ist die feindliche Nutzung oder der Versuch der feindlichen Nutzung einer API. Im Folgenden sind einige der vielen Möglichkeiten aufgeführt, wie Angreifer einen API-Endpunkt missbrauchen kann. Die Liste ist nicht vollständig.

Brut-Force / Credential Stuffing

Brute-Force-Angriffe versuchen, Passwörter ohne Kontext oder Anhaltspunkte zu erraten, indem sie alle möglichen Kombinationen von Zeichen ausprobieren. Credential Stuffing verwendet exponierte Anmeldedaten, wodurch die Anzahl der möglichen richtigen Antworten drastisch reduziert wird. Gegen einen Brute-Force Angriff ist ein langes Kennwort das Grossbuchstaben, Zahlen und Sonderzeichen enthält ein guter Schutz. Auf einen Credential Stuffing Angriff hat es keine Wirkung. Credential Stuffing macht gemäss dem Bericht von Akamai (vgl. [22]) benötigen Kriminelle drei Dinge, um Credential-Stuffing- Angriff durchzuführen:

„*Zunächst brauchen sie Zugangsdatenlisten (auch Kombinationslisten genannt), die Nutzernamen und Kennwörter enthalten. Weiterhin benötigen Kriminelle ein bekanntes Ziel mit zugänglichen Authentifizierungsmechanismen. In vielen Fällen handelt es sich bei diesen Authentifizierungsmechanismen um API-Endpunkte. Schließlich brauchen sie ein Tool, um diese Listen sowie Konfigurationsdateien zu nutzen und ordnungsgemäss mit dem Authentifizierungsmechanismus des Opfers zu kommunizieren“* [22]

Aufgrund der zahlreichen Bots ist Credential Stuffing zu einer beliebten Angriffsmethode geworden. Anmeldungs-Endpoints schützen sich oft durch das Sperren von IP-Adressen oder einem zeitlich begrenzten Rate-Limit.

Moderne Credential Stuffing Angriffe umgehen diese Schutzmassnahmen mit Bots, die gleichzeitig mehrere Anmeldeversuche durchführen, die von verschiedenen IP-Adressen und unterschiedlichen Gerätetypen stammen. Die Bots versuchen den Angriff so durchzuführen, dass er von einem typischen Login Zugriff nicht unterscheidbar ist, was sehr wirksam ist (vgl. [22])

Injection Angriffe

Von Injektionsangriffe spricht man, wenn schädlicher Code in die Applikation eingeschleust wird. SQL Injections (SQLi) und Cross-Site Scripting (XSS) Angriffe sind die bekanntesten Beispiele. Injektionsangriffe sind seit langem eine Bedrohung für Webanwendungen und auch eine Gefahr bei APIs.

DoS / DDoS Angriffe

Bei einer Denial of Service (DoS)- oder Distributed Denial of Service (DDoS)-Attacke versucht der Angreifer, ein System für die Benutzer unerreichbar zu machen. Sie haben eine grosse Bandbreite an möglichen Ausmassen. DoS-Angriffe werden oft mit DDoS-Angriffen (Distributed Denial of Service) verwechselt, sind aber einfacher abzuwehren.

Bei DoS-Angriffen richtet ein bösartiger Angreifer einen einzigen Roboter auf die API und stellt eine Reihe von wiederholten Anfragen an einen Endpunkt für eine beträchtliche Zeitspanne mit einer sehr hohen Frequenz. Die Flut von Anfragen übersteigt die Reaktionsfähigkeit der API, wodurch sie nicht mehr in der Lage ist zu reagieren. Um solche Roboter zu blockieren, kann man mit grundlegenden Funktionen einer WAF, wie IP und GEO-Blocking arbeiten (vgl. [24]). Es gibt noch weitere Massnahmen wie das Blockieren bestimmter Klassen von User Agents (z. B. Browsertypen) oder die elastische

Kapazität von Cloud-Servern, um der höheren Belastung entgegenzuwirken. Mittels solcher Massnahmen kann ein Dos-Angreifer erfolgreich abgewehrt werden.

Das Problem dabei ist das Kriminelle ihre Angriffsmethoden anpassen und merken wenn sie blockiert wurden und dann andere Angriffe, wie beispielweise eine DDoS Attacke ausprobieren.

Ein DDoS Attacke unterscheidet sich von einer DoS Attacke dadurch, dass sie aus mehreren Robotern besteht, die von verschiedenen IP-Adressen und Standorten aus ihren Angriffen koordinieren und somit die IP-Sperrung umgehen. Meistens setzt der Angreifer VPNs ein und kann damit die IP-Adressen der Roboterarmee ständig verändern. Geografische Sperren können hier bis zu einem gewissen Grad funktionieren, haben aber immer noch einige Unzulänglichkeiten, wenn es um DDoS-Schutz geht:

Wenn ein echter DDoS Angriff erfolgt, dann gibt es nicht viel, was man tun kann, ausser zusätzliche Ressourcen für den Cloud-Server anzufordern und die Attacke so möglichst gut überstehen. DDoS-Angriffe werden in der Regel nicht sehr lange aufrechterhalten, da sie im Durschnitt weniger als vier Stunden dauern (vgl. [25])). Die Performance des Servers ist hier entscheidend. Wenn der Server schon mit wenigen Anfragen überfordert ist, kann eine DDos-Attacke natürlich sehr viel länger und auch kostengünstiger durchgeführt werden.

Buffer-Overflow Angriff

Buffer-Overflow-Angriffe versuchen, ein System auszunutzen, indem sie

indem sie ihm Daten ausserhalb des erwarteten Bereichs oder Typs liefern, was zu Systemabstürzen führen kann oder den Zugriff auf eine Speicherstelle zulässt. Der Angriff wurde klassisch gegen C-Programme angewendet und war früher sehr beliebt. Er existiert aber auch noch heute.

Parameter Manipulation

Ein Angreifer versucht, die zwischen Client-Applikation und API ausgetauschten Parameter zu manipulieren. Ziel ist es, Daten zu verändern, wie beispielsweise Benutzerberechtigungen, oder den Preis einer Bestellung. Parameterangriffe nutzen die gesendeten Daten an eine API, einschließlich URL, Abfrageparameter, HTTP-Header

Oder Body-Inhalte von Post Requests. Der klassische SQL Injection Angriff wird fast immer aufgrund einer erfolgreichen Parameter Modifikation ermöglicht. Auch Cross-Site Scripting (XSS) Attacken gehören zu diesem Angriff. Das

klassisches Beispiel für eine XSS-Attacke, ist ein JavaScript-Code, der in einem Beitrag auf einem Online-Forum gespeichert wird. Das Skript wird von jedem nachfolgenden User, der den Beitrag liest, ausgeführt. Das Skript wird in seinem Browser ausgeführt, wodurch möglicherweise das Access Token oder andere sensible Informationen ausgelesen werden.

Man in the Middle (MitM)

Bei einem MitM-Angriff fängt der Angreifer die Kommunikation zwischen einem API-Endpunkt und einem Client ab. Der Angreifer stiehlt oder verändert die vertraulichen Daten, die übermittelt werden. Wenn eine API SSL/TLS nicht überall korrekt verwendet,

können die Anfragen und Antworten zwischen einem Client und dem API-Server kompromittiert werden.

Schlussfolgerung Die oben aufgeführten Angriffe sind in allen Branchen zu beobachten. Darüber hinaus gibt es aber auch Angriffe, die spezifischer sind und auf die jeweilige API zugeschnitten. Dazu werden die Schwachstellen einer API ausgenutzt, welche wir im nachfolgenden Kapitel genauer betrachten. Laut dem am 3.Feb 2021 veröffentlichten Bericht von Salt Security (vgl. [23]) sind 54% der Sicherheitsprobleme auf Schwachstellen in der API zurückzuführen. Im nachfolgenden Kapitel werden wir deshalb auf die Schwachstellen von APIs eingehen und diese auf ihre Testbarkeit analysieren.

1.6.3.4 API Security Testing

Was sind API Tests

API-Tests sind eine Art von Tests, bei denen APIs direkt und als Teil von Integrationstests getestet werden, um ihre korrekte Funktionalität und Sicherheit zu überprüfen. Im Gegensatz zum Testen einer Webapplikationen, das sich auf das UI konzentriert, findet API-Tests in der Business-Schicht der Applikation statt. Diese Schicht wird auch Nachrichtenebene genannt, da dort ist für den End-User normalerweise unsichtbar, da APIs keine grafischen Benutzeroberflächen besitzen. (vgl. [26])

API-Tests sind besser geeignet für Automatisierte Tests als die Benutzeroberfläche. Die Benutzeroberfläche (UI) zu testen ist sehr aufwendig, da mit automatisierten Browsern gearbeitet wird, welche das jeweilige UI untersuchen. Das Problem hierbei ist, dass sich die Benutzeroberfläche in der heutigen Zeit schnell verändert. Durch die Veränderungen werden eine Vielzahl von Tests fehlschlagen und müssen entsprechend modifiziert werden. Der entstandene Aufwand, um alle Tests auf dem aktuellen Stand zu halten, erhöht sich damit massiv.

API-Tests sind für Automatisierungstests besser geeignet, da sie weniger häufige Änderungen erfahren und mit den kurzen Release-Zyklen der Benutzeroberfläche zurechtkommen (vgl. [26]). Dadurch erfordern die Tests weniger Wartungsaufwand im Vergleich zu UI-Automatisierungstests, was sie zu einer bevorzugten Wahl macht.

Wo werden API Tests durchgeführt

Die Web-Entwicklung ist von Natur aus in Schichten aufgeteilt. Es verwendet das Client-Server-Modell und es gibt eine Vielzahl an Protokollen. Daher ist es naheliegend, dass eine Schichtenarchitektur für die Entwicklung für das Web geeignet ist. Üblicherweise werden Web-Applikationen in drei Schichten unterteilt (vgl. [26]) :

- Der Presentation-Layer ist für den Benutzer in Form einer grafischen Oberfläche sichtbar und über den Browser zugänglich. Die Schicht besteht aus drei Kerntechnologien, HTML, CSS und JavaScript. Auf dieser Schicht verwenden Entwickler beispielsweise Angular, um eine Seite dynamisch zu aktualisieren.
- Der Business-Layer beinhaltet die Geschäftslogik und ist über die API erreichbar. Die Schicht nimmt Benutzeranfragen vom Client entgegen, verarbeitet sie und liefert schlussendlich eine Antwort. Ein API-Endpoints hat immer eine bestimmte Funktion oder Rolle, welche er erfüllt. Gemäss den Microservice API-Patterns (vgl. [27]) besitzt ein End Point immer eine bestimmte Rolle. Entweder ist er eine „Information Holder Ressource“, um für eine Anfrage Daten liefern zu können oder eine „Processing Ressource“, die beispielsweise Objekte modifiziert oder Berechnung durchführt
- Der Data-Layer ist ein zentraler Ort, der alle Datenaufrufe von der Business-Schicht empfängt. Hier werden alle Daten der Webapplikation persistent gespeichert.

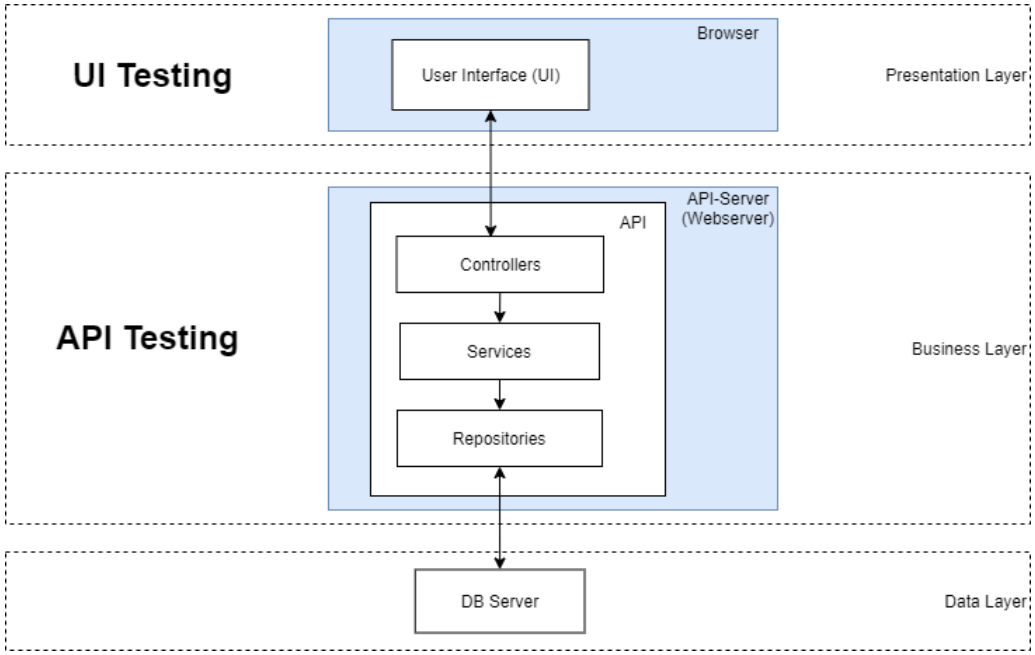


Abbildung 14: Schichten einer Webapplikation

Automatisierung von API Tests

Die Pyramide der Automatisierungstests ist eine grafische Darstellung, die helfen soll, eine besser automatisierte Teststrategie zu entwickeln. Die Schichten repräsentieren verschiedene Arten des Testens. Sie sind stark vereinfacht und die wesentliche Aussage ist, dass man viel mehr Unit- und Integration-Tests haben soll, da man mit der investierten Zeit den höchsten Rückgabewert erzielt. (vgl. [26])

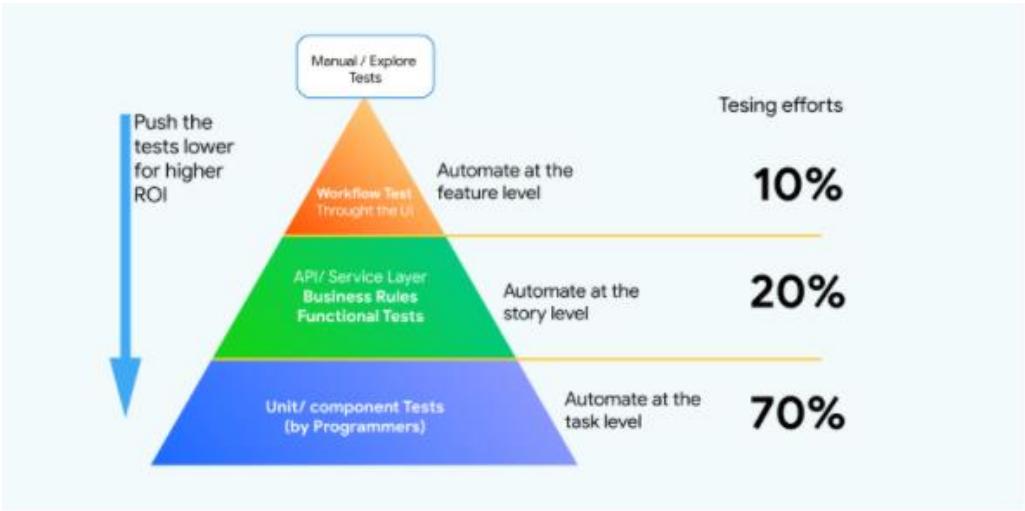


Abbildung 15: Testpyramide von Katalon

Die Unit- und Integration Tests können vollständig automatisiert werden. Man sollte etwa 70% des Testaufwands in Unit- und Integrations-Tests aufwenden, um eine möglichst hohe Codeabdeckung zu erreichen. Die API Tests befinden sich in der zweiten Ebene und testen die korrekte Funktionsweise der API gegen ihre User Stories. Die Tests können ebenfalls zu einem hohen Grad automatisiert werden und stellen sicher das die API wie erwartet funktioniert. Sie sind UI-Tests vorziehen. Bei der Automatisierung von Tests sollte man sich am wenigsten auf UI Tests konzentrieren, da die Tests durch jede Änderung die Tests abbrechen kann (vgl. [26]). Abschliessend kann gesagt werden das

API-Tests in der zweiten Schichte eine wichtige Aktivität darstellen, auf welche wir uns konzentrieren müssen.

Security Testing

Sicherheits-, Penetrations- und Fuzz-Tests sind die drei separaten Typen von API Testing. Sie werden eingesetzt, um sicherzustellen, dass die API-Implementierung vor externen Bedrohungen sicher ist (vgl. [26])

Security Tests sind als White-Box Tests zu verstehen, welche von der jeweiligen Organisation selbst durchgeführt werden. Es werden die Sicherheitsanforderungen überprüft. Es wird also geprüft, ob die Authentifizierung korrekt funktioniert. Wie gut sensible Daten verschlüsselt sind. Tests, welche den Code auf Schwachstellen überprüfen gehören ebenfalls in diese Kategorie. Die grundlegenden funktionalen Tests von API ist einfach, wenn man immer vom den richtigen Inputdaten ausgeht. Beim Security Testing geht es darum, die Sicht eines Angreifers anzunehmen. Bei der Suche nach Schwachstellen wird er die Inputdaten modifizieren und überprüfen, wie die API darauf reagiert. Um die Sicherheit einer API zu gewährleisten ist es deshalb entscheidend, die API auch auf unerwartete Eingaben zu testen. Es können deshalb die folgenden einfachen Regeln für das Testen einer API definiert werden:

- Für einen bestimmten Input muss die API eine erwartete Ausgabe liefern
- Eingaben eines falschen Typs müssen zurückgewiesen werden
- Jede Eingabe, die null (leer) ist, muss zurückgewiesen werden, wenn dies nicht akzeptabel ist
- Inputdaten mit einer falschen Grösse oder einem zu grossem Wert müssen verworfen werden

Penetration Tests

Penetration Tests werden üblicherweise ausserhalb einer Organisation durchgeführt und ein Benutzer mit begrenzten API-Kenntnissen versucht einen Angriff zu starten. Diese Tests können ebenfalls mit einem Tool umgesetzt werden. Jedoch sind die Möglichkeiten dieser Tools eingeschränkt und Firmen setzten oft Experten ein um manuelle Penetrationtests durchzuführen.

Fuzz Testing

Von Fuzz Testing spricht man wenn eine grosse Menge an Anfragen mit zufälligen Daten (als „Fuzz“ oder Rauschen bezeichnet) an die API gesendet werden. Das Ziel ist es den Webserver zum Absturz zu bringen oder ein negatives Verhalten festzustellen. Damit werden die Grenzen einer API ausgetestet, um festzustellen ab welcher Verkehrslast es zu einem Einbruch kommt (vgl. [26])

1.7 Lösungskonzept

Konzeptphase

Im Lösungskonzept wird die Microservice-Architektur für die API bestimmt und genauer beschrieben. Die gesammelten Informationen aus den Security Grundlagen und der Tool-Evaluation sind in das Lösungskonzept miteinbezogen worden. Diese Informationen waren die Grundlage für die wichtigsten Entscheidungen.

1.7.1 Ziele definieren

Einleitung

Klare Ziele machen den gesamten Prozess straffer und konzentrieren sich nur auf die relevanten Objekte. Ziele sind auch notwendig, um die relevanten Sicherheits-Anforderungen für das Testen der API zu definieren.

Zweck der API

Die Trading-API erlaubt es Kunden, Aufträge zum Kauf oder Verkauf von Wertschriften (Assets) von Zuhause aus durchzuführen. Neue Kunden werden durch einen Administrator hinzugefügt, indem er einen neuen Account erstellt. Nur er kann eine platzierte Order entfernen oder ausführen. Kunden authentifizieren sich über den API Gateway an einem Service für die Authentifizierung. Mit einem gültig signierten Token, erhalten sie den Zugriff auf die API.

Idee / Konzept

Da es sich bei einem Application Programming Interface (API) um ein Informationssystem handelt können die allgemeinen Schutzziele der Informationssicherheit auch für die Entwicklung und das Design der API angewendet werden. Die Informationssicherheit setzt sich aus drei wesentlichen Grundbedrohungen zusammen: Die Vertraulichkeit, die Integrität und die Verfügbarkeit. Durch diese Schutzziele kann sichergestellt werden, wie weit die Systeme eines Unternehmens die Informationssicherheit erreicht haben. Wenn die Schutzziele erfüllt sind, ist die Applikation Werden die Schutzziele erfüllt, ist das System eines Unternehmens gegen unbefugte Einwirkungen und Angriffe geschützt.

Vertraulichkeit:

Mit diesem Ziel soll sichergestellt werden, dass Daten nur für autorisierte Benutzer verfügbar sein dürfen. Die Informationen müssen zu jedem Zeitpunkt geschützt sein. Es ist zum einen die Sicherheit von gespeicherten Daten gemeint und zum anderen die Sicherheit der Informationen bei der Datenübertagung.

Integrität:

Integrität ist die Zusicherung, dass die Informationen korrekt ist und zu keinem Zeitpunkt unautorisiert verändert wurde. Bei einer unzulässigen Änderung soll diese zuverlässig Erkannt werden.

Verfügbarkeit:

Dieses Ziel bezieht sich auf die tatsächliche Verfügbarkeit der Daten. Es wird sichergestellt, dass die Daten für autorisierte Personen verfügbar sind, wenn sie benötigt werden.

Neben den allgemeinen Schutzzielen wurden mit dem Praxispartner zusammen noch weitere nachfolgende Ziele definiert:

- **Frühzeitige Erkennung von Schwachstellen**, um diese rechtzeitig schliessen zu können
- **Simulieren von Angriffen**, um so nicht geschlossene Schwachstellen zu finden
- **Testen der korrekten Funktionalität.** Neben Integrationstests sollen auch End-to-end Test mit einem Tool durchgeführt werden**,** um das Risiko eines Sicherheitsproblems zu minimieren
- **Keine Speicherung von Personenbezogenen Daten**.

1.7.2 Sicherheitsanforderungen definieren

Einleitung Die oben definierten Ziele müssen durch entsprechende Security Anforderungen messbar und kontrollierbar sein.

Idee / Konzept Aus den definierten Zielen lassen sich die nachfolgenden Security Anforderungen definieren:

Ziele	Security Anforderung oder Kontrolle
Vertraulichkeit: Accounts eines Kunden dürfen nur für diesen sichtbar sein und müssen entsprechend gegen einen nicht autorisierten Zugriff geschützt sein. Admin-Funktionen dürfen nur entsprechend berechtigen Personen zugänglich sein. Alle anderen Benutzer, welche keinen gültigen Zugriff auf die entsprechenden Nutzerdaten besitzen, dürfen keinen Zugriff auf die Informationen erhalten.	Einhaltung der strengen Kontrolle der Anwendungsarchitektur in Bezug auf Authentifizierung und Zugriffskontrolle. Einsatz von JWT Tokens und Anwendung des OAuth2 Standard in Verbindung mit Open ID Connect wird vorausgesetzt. Die Authentifizierung erfolgt global über dem Gateway an einem Identity Management Service. Die Verbindung muss mit TLS jederzeit verschlüsselt erfolgen. Der erstellte Token darf nur für die entsprechende Applikationen gültig sein. Es darf kein Token ausgestellt werden, welcher für mehrere Applikationen verwendet werden kann. Ein Token darf maximal 5min lang gültig sein, danach muss ein neues Token ausgestellt werden.
Integrität: Es darf nicht möglich sein das eine Meldung auf dem Weg verändert wird z.B. auf dem Weg von Gateway zur entsprechenden API. Das Risiko eines Man in the Middle Angriffs soll minimiert werden. Ein Auditing von kritischen Funktionen und Datenänderungen gehört ebenfalls zur Sicherung der Integrität	Korrekte Implementation von aktuellem TLS Standard. Verwendung von Security Headern wie HSTS, welcher eine HTTPS Verbindung erzwingt und so SSL-Stripping Angriffe (vgl. [28]) effektiv verhindert. Eine Content-Security-Policy (CSP) wird eingesetzt um mögliche AÄnderungen in der API müssen nachvollziehbar sein und durch eine Versionierung zu jeder Zeit nachvollzogen werden.
Verfügbarkeit: Die API muss zuverlässig abrufbar und nutzbar sein für diejenigen, welche ein Zugriffsrecht haben. DOS-Angriffe müssen durch entsprechende Methoden verhindert werden.	Einsatz eines API-Gateways, welches als primäres Kontrollorgan den Traffic überwachen und mittels Policy auch begrenzen können. Es muss für jede API individuell bestimmt werden, welcher Traffic zu erwarten ist und ab wo eine Grenze gezogen wird. Die Web Applikation Firewall (WAF) eignet sich zum Sperren von IP-Adressen und GEO-Blocking

Ziele	Security Anforderung oder Kontrolle
Frühzeitige Erkennung von Schwachstellen, um diese rechtzeitig schliessen zu können	Einsatz von Static Application Security Testing (SAST). Gefundene Schwachstellen mit einem zu hohen Score müssen genauer geprüft. Report wird als Ergebnis erwartet
Simulieren von Angriffen, um so nicht geschlossene Schwachstellen zu finden	Einsatz von Dynamic Application Security Testing (DAST).
Testen der korrekten Funktionalität und prüfen von API-Schwachstellen mit Tests, um das Risiko eines Sicherheitsproblems zu minimieren	Bei jedem Deployment der Applikation müssen Test-Reports erstellt und zur Verfügung gestellt werden.
Keine Speicherung von Personenbezogenen Daten.	Personenbezogene Daten müssen immer verschlüsselt übertragen und gespeichert werden. Für die Verschlüsselung wird AES-256 verwendet In der API werden keine bewusst keine personenbezogenen Daten gespeichert da diese bereits auf anderen Services vorhanden sind.

1.7.3 Architektur definieren

Einleitung

Nachdem die Sicherheitsanforderungen geklärt worden sind, musste als nächstes eine Architektur definiert werden, um die Sicherheitsanforderungen erfüllen zu können. Diese definiert gleichzeitig auch die Angriffsfläche, welche analysiert wird. Durch die Technischen Komponenten wird definiert, welche Software und Bibliotheken bei der Arbeit zur Anwendung kommen. Das nachfolgende Konzept wurde in Absprache mit dem Praxispartner erstellt.

Idee / Konzept

Globale Authentifizierung über Gateway + Lokale Service Autorisierung

Die Regelung der Zugriffskontrolle ist einer der wichtigsten Architekturentscheide, wenn es um das Thema Sicherheit geht. Es wurde vereinbart für das Ausstellen eines JWT Tokens, den Identity und Access Management Provider Keycloak [29] zu verwenden. Mit Keycloak kann der OAuth2 Standard in Verbindung mit Open ID Connect realisiert werden. OAuth ist ein weit verbreiteter Sicherheitsstandard, der den sicheren Zugriff auf geschützte Ressourcen einer API ermöglicht. Er ist besonders für Microservice-Architekturen geeignet. Als API Gateway wird Tyk [30] eingesetzt. Der API Gateway authentifiziert sich über Open ID Connect beim Keycloak-Server und erhält bei erfolgreicher Prüfung ein OAuth2 konformes JWT Token.

Der API Gateway eignet sich besonders für die Authentifizierung von Benutzern, da er einen einzelnen Zugriffspunkt für den Benutzer darstellt. Über ihn werden Authentifizierungsanfragen gestellt, welche er dann an den Keycloak-Service weiterreicht. Damit wird garantiert, dass alle Microservices denselben Authentifizierungsprozess implementieren. Wenn sich ein Benutzer erfolgreich authentifiziert hat, reicht der Tyk Gateway die Anfrage des Users mit dem Access Token an und leitet sie an die entsprechenden Service. Der Service selbst, also die API, führt dann die Zugriffsprüfung durch.

Vorteile

- Kein Single Point of Failure für die Autorisierung
- Feinkörnige Berechtigungen möglich, da die API selbst entscheidet, worauf ein Benutzer Zugriff hat.
- Autorisierung wird durch jeweiligen Microservice (API) gesteuert und nicht durch den API Gateway, damit wird die Last aufs Netzwerk reduziert und die Latenz besser sein.

Nachteile

- Single Point of Failure für Authentifizierung.
- Erhöhter Aufwand für Entwickler, da mehr Code nötig und er sich mit der Berechtigung vertraut machen muss.

Spring Security

Für die Umsetzung der lokalen Autorisierung in der API wird Spring Security verwendet. Mit Spring Security ist es möglich die entsprechenden Ressourcen im Kontext des OAuth-Tokens (JWT Token), welcher vom Gateway geleifert wird, zu schützen. Die Aufgabe der API ist es das Token zu validieren, bevor der Zugriff auf eine Ressource gewährt wird.

Einsatz von Zed Attack Proxy (ZAP)

Ein Dynamic Analysis Security Testing Tool (DAST-Tool) hilft dabei bestimmte Schwachstellen in produktiven Webapplikationen oder APIs zu finden. Ein DAST-Test wird auch als Black-Box-Test bezeichnet, da der Tester keinen Einblick in den internen hat, er verwendet im Wesentlichen die gleichen Techniken, die ein Angreifer verwenden

würde, um potenzielle Schwachstellen zu finden. Ein DAST-Test kann nach einer breiten Palette von Schwachstellen suchen, einschließlich Eingabe-/Ausgabe-Validierungsproblemen, die eine Anwendung anfällig für Cross-Site-Scripting oder SQL-Injection machen könnten. Ein DAST-Test kann auch dabei helfen, Konfigurationsfehler zu erkennen und andere spezifische Probleme der Applikation zu identifizieren. Der Zed Attack Proxy [31] von OWASP ist ein weitverbreitetes DAST-Tool und wird bei dieser Arbeit eingesetzt.

Code Security

Das statische Testen der Sicherheit (SAST) ist eine White-Box-Testmethode, welche den Quellcode und die Binärdateien einer Applikation bewertet. Damit können potenzielle Sicherheitsschwachstellen frühzeitig identifiziert und geschlossen werden. Mit dem Praxispartner wurde vereinbart Snyk [32] zur Erkennung und Behebung von Schwachstellen im Code einzusetzen. Snyk verfügt über eine grosse Datenbank von Schwachstellen und geht über CVE-Schwachstellen und andere öffentliche Datenbanken hinaus.

Automatisierte Unit- und Integration-Tests

Die automatisierten Unit- und Integration-Tests werden mit JUnit 5 (vgl. [33]) geschrieben. Die benötigten Mocks werden mit Spring MVC umgesetzt. Die Test Coverage wird mit Jacoco (vgl. [34]) ausgewertet. Das festgelegte Ziel ist es, eine Test Coverage von mindestens 60% zu erreichen. Die Test Coverage wird in der CI/CD Pipeline automatisch angezeigt und der neu erstellte Code muss immer alle Unit- und Integration-Tests erfolgreich bestehen.

End-to-End Security Testing

Für die End-to-End Tests wurde das Tool Postman evaluiert. Mit Postman sollen die Schwachstellen einer API und die korrekte Authentifizierung und Autorisierung getestet werden. Die Tests sollen automatisch von der CI/CD Pipeline ausgeführt werden sobald die API auf die Produktion ausgeliefert wurde.

Web Application Firewall (WAF)

WAFs können schädlichen Datenverkehr blockieren, bevor er an den API-Gateway weitergeleitet wird, indem sie sofort einsetzbare Funktionen wie IP-Blocking, Angriffssignaturen, Erkennung bösartiger Bots und OWASP Top 10 Schwachstellenerkennung für Web-Applikationen verwenden. Der Einsatz einer WAF ist zwingend notwendig, um sich vor bestimmten Angriffen wie z.B. einer DOS-Attacke schützen zu können. Es wurde mit dem Praxispartner entschieden das die Behandlung der WAF in dieser Arbeit nicht weiter vertieft werden soll. Sie ist deshalb kein Bestandteil der Arbeit

Architektur Aus dem definierten Konzept ergibt sich die in der Abbildung ersichtliche Architektur.

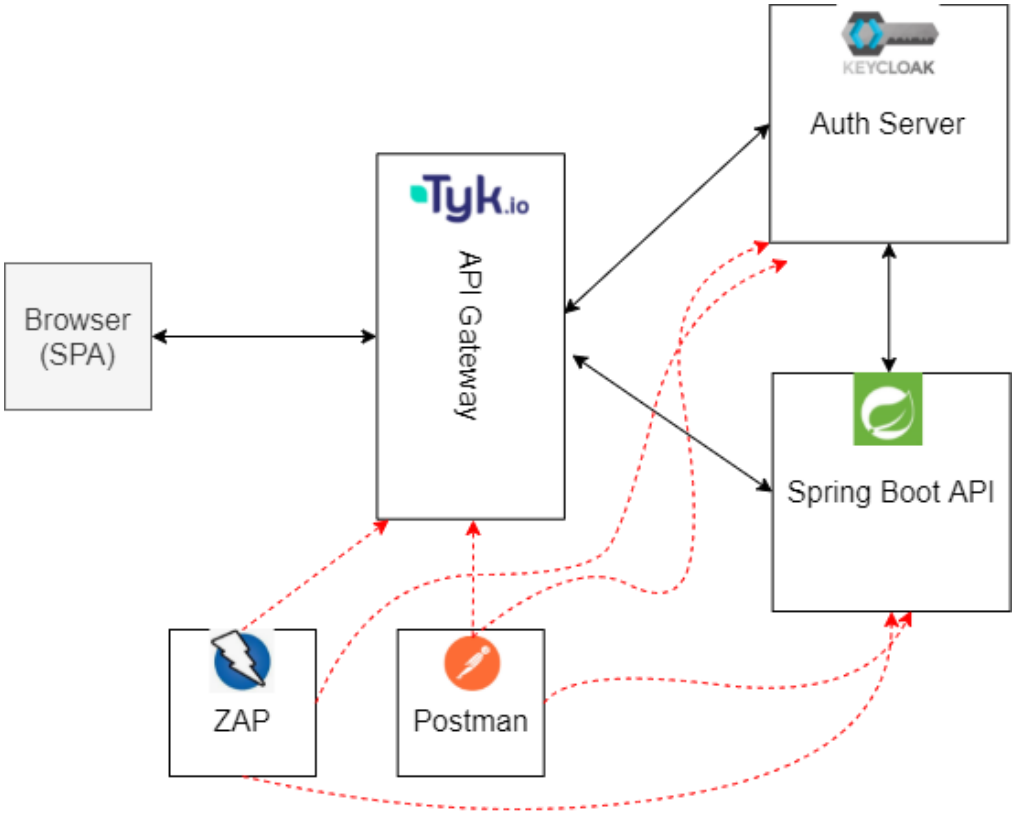


Abbildung 16: Architektur API Security Testing

Die Microservice-Architektur besteht aus dem API Gateway „Tyk“ , dem Authentifikation-Service „Keycloak“ und der Spring Boot API. Die verschiedenen Services werden von aussen mit Postman und dem Zed Attack Proxy (ZAP) auf ihre Schwachstellen überprüft.

1.7.4 Bedrohungsmodell

Einleitung

Mit den definierten Zielen wurde festgelegt, was wichtig ist, um daraus Sicherheitsanforderung zu generieren. Danach konnten die Technischen Komponenten bestimmte werden. Durch die Bedrohungs-Modellierung soll nun aufgezeigt werden, wie die bestimmten Komponenten miteinander sprechen, um so potenzielle Schwachstellen zu identifizieren und geeignete Schutzmassnahmen zu priorisieren. Eine Organisation kann sich niemals gegen alle Risiken schützen, weshalb eine Priorisierung von Massnahmen festgelegt werden muss. Da die Arbeit sich vertieft mit dem Testing einer API beschäftigt, ist es wichtig festzulegen, welche Schwachstellen und Massnahmen durch Security Tests überprüft werden.

STRIDE

STRIDE ist das Urgestein der Bedrohungsmodellierung, das im Jahr 1999 bei Microsoft entwickelt wurde. STRIDE steht für die sechs Bedrohungskategorien. Für die Einteilung der OWASP Top 10 Schwachstellen in diese Kategorien wird das Bedrohungsmodell STRIDE verwendet. Im nachfolgenden werden deshalb die verschiedenen Kategorien von STRIDE aufgelistet und kurz erläutert:

Spoofing (Vortäuschung)

Da eine API den Zugriff auf ein System oder Daten erlaubt, muss für jeden Zugriff die Identität des Gegenübers sichergestellt werden. Sicherheit der API hängt massegebend vom Vertrauen der Identität ab. Wenn dieses Vertrauen gebrochen wird sprich man von Spoofing. Man gibt vor jemand zu sein, der man nicht ist. Sowohl die Identität des Betreibers wie auch die des Konsumenten können gefälscht werden. (vgl. [35])

Tampering (Manipulation)

Wenn man als API Client oder als API Provider Daten von einem System bezieht muss man sich darauf verlassen, dass diese zuverlässig sind und nicht manipuliert wurden. Daten sind besonders anfällig für Manipulationen, aber auch physische und virtuelle Server können anfällig sein. Oft geht die Manipulation von Daten mit anderen Bedrohungen einher. Zum Beispiel können Datenmanipulationen durch fälschlich erhöhte Berechtigungen verursacht werden oder Daten werden verändert, um beim Zugriff eine andere Identität vorzuspielen.

Repudiation (Abstreitbarkeit)

Mit Repudiation ist die Behauptung gemeint, dass der Benutzer etwas nicht getan hat oder nicht dafür verantwortlich gemacht werden kann. Angriffe auf ein System können nicht verhindert werden, aber man kann Audits implementieren, um bestimmte Aktivitäten verfolgen zu können. Bei richtiger Anwendung können sie durch die Protokollierung Angreifer lokalisieren und so beispielsweise Brute-Force Angriffe frühzeitig erkennen und unterbinden. Sichere Systeme sollten einen nicht abstreitbaren Mechanismus einbauen, damit der Datenquelle und deren Daten selbst vertraut werden kann. (vgl. [35])

Information Disclosure (Offenlegung)

Information Disclosure bedeutet die Weitergabe von Informationen an nicht berechtigte Personen. Informationen können während der Kommunikation oder während der Speicherung auf dem Rechner des Benutzers oder des Betreibers durchsickern. Bedeutende Datenschutzverletzungen haben in den letzten Jahren für Schlagzeilen gesorgt. Passwörter, Kreditkarten Informationen und andere persönliche Daten kommen immer wieder in die falschen Hände. Neben Bedrohungen wie Spoofing können

Offenlegungen auch durch Backups und ungeschützte alte Versionen versehentlich erfolgen.

Denial of Service (Verweigerung von Diensten)

Denial-of-Service-Angriffe versuchen, die Services zu stören, indem sie die Leitung für die Kommunikation überlasten oder im schlimmsten Fall den Absturz eines Systems zu erzwingen. Bei APIs bedeutet dies, dass die API mit eingehenden Anfragen überlastet wird, so dass anderen Konsumenten unmöglich ist eine Verbindung herzustellen.

Elevation of Privilege (Erhöhung der Privilegien)

Elevation of Privilege bedeutet eine Erhöhung der Berechtigung und bezieht sich auf die Autorisierung. Wenn es dem Angreifer beispielsweise möglich ist seinen Benutzer mit erweiterten Rechten auszustatten.

Ein erfolgreicher "Elevation of Privilege"-Angriff kann alle anderen Bereiche von STRIDE verletzten und erzielt so einen besonders hohen Schaden. Mit erhöhtem Rechten kann der Angreifer möglicherweise auf Umsysteme der API zugreifen. (vgl. [35])

Abgrenzung Schwachstellen

Diese Bedrohungsmodellierung konzentriert sich bei dieser Arbeit auf das Testing der OWASP API Security Top 10 Schwachstellen. Da im Rahmen der Arbeit nicht alle Schwachstellen getestet werden können muss eine Priorisierung erfolgen. Einige der OWASP Top 10 API-Schwachstellen haben sich durch die Verteilung der Aufgaben auf andere Services verschoben.

Priorisierung

Für die Priorisierung werden nachfolgende Prioritäten vergeben:

- Priorität 1 ➔ Schwachstelle ist testbar und betrifft die Trading API-Komponente.
- Priorität 2 ➔ Schwachstelle ist testbar aber betrifft nicht die Trading-API vorhanden.
- Priorität 3 ➔ Schwachstelle ist nicht testbar

Für die Priorisierung wurde eine Gewichtung von 1 bis 3 durchgeführt. Die Zahl 1 hat die höchste Priorität, und soll in dieser Arbeit getestet werden. Die Zahl 2 hat die mittlere Priorität und wird bei genügend Zeitreserven noch behandelt. Priorität 2 wird implementiert, sobald alle Test-Cases mit der Priorität 1 umgesetzt wurden. Alle Test-Cases mit Priorität 3 können nicht getestet werden und sind deshalb nicht Teil der Arbeit.

Bedrohungsmodell

Für das Erstellen des Daten-Flow Diagramms wurde OWASP Threat Dragon (vgl. [36]) verwendet. Es zeigt auf die einzelnen Komponenten miteinander kommunizieren. Damit können potenzielle Bedrohungen in der Zukunft.

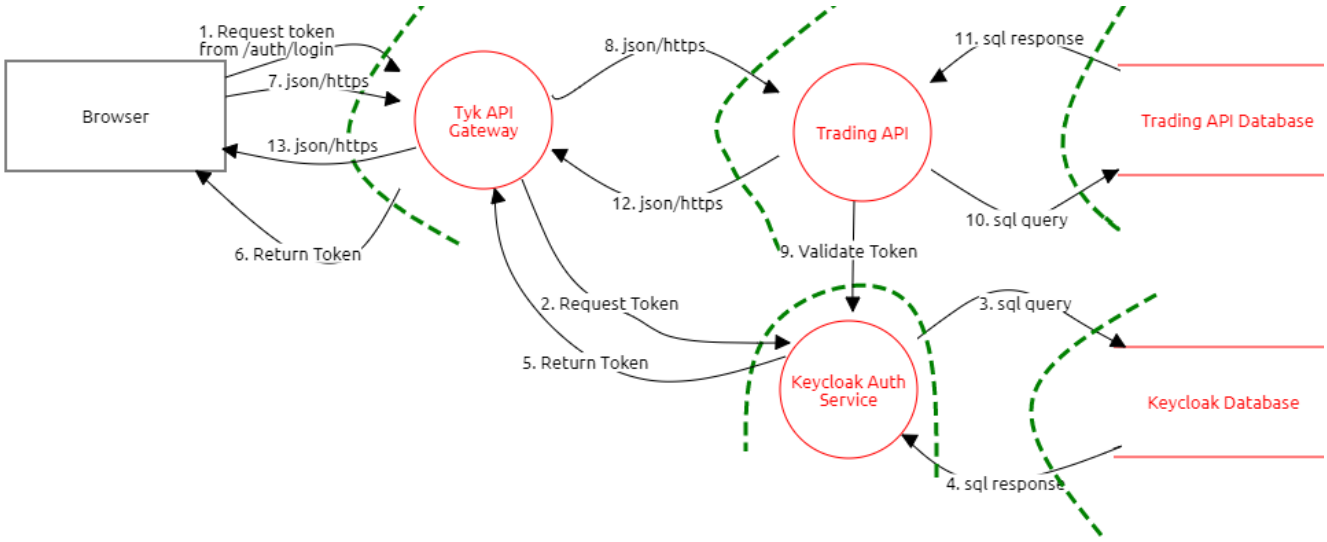


Abbildung 17: Bedrohungsmodell erstellt mit Threat Dragon

Das erstellte Modell zeigt den Data-Flow des Clients an wenn er sich mit einer Ressource der Trading API verbindet.

1. Um sich zu identifizieren stellt ein Client eine Anfrage an die API Gateway URL /auth/login
2. Der Gateway leitet die Anfrage an den Keycloak Authentification Service weiter
3. Der Keycloak Service prüft die Zugangsdaten in seiner Datenbank
4. Die Keycloak Datenbank teilt dem Service das Ergebnis mit.
5. Wenn die Zugangsdaten korrekt sind wird JWT Token vom Keycloak-Service an den API Gateway zurückgeben
6. Der API Gateway teilt das JWT Token und erstellt ein Key-Value Paar, aus der Signatur des JWT Tokens. Dieser Key wird dem Client zurückgegeben, um sich an der API zu authentifizieren.Der Header und Body des JWT Tokens werden nicht an den Client herausgegeben.
7. Mit dem erhaltenen Opaque Token stellt der Client eine Anfrage an den API-Gateway, um auf eine Ressource der Trading-API zuzugreifen.
8. Der Gateway prüft in seiner Datenbank, ob er für den erhaltenen Opaque Token ein entsprechendes JWT besitzt. Wenn ein JWT gefunden wurdem wird die Client-Anfrage mit dem korrekten JWT an die dahinterliegende API weitergeleitet.
9. Die API validiert das Token und überprüft, ob das Token autorisiert ist um die entsprechende Aktion auszuführen.
10. Bei erfolgreicher Überprüfung setzt die API eine Datenbankabfrage ab um dem Client die gewünschten Informationen zu lieferen
11. Die Datenbank beantwortet die Anfrage der API und teilt ihr das Resultat mit.
12. Die API liefert dem Gateway die Antwort in Form eines JSON Objektes.
13. Der Gateway leitet die Antwort weiter an den Client

CVSS

Normalerweise wird für die Bewertung der gefunden Schwachstellen das Common Vulnerability Scoring System (auch bekannt als CVSS-Scores) vom National Institute of Standards and Technology (NIST) verwendet (vgl. [37]). Dieses System stuft die Wichtigkeit von Schwachstellen nach dem CVSS Score ein:

Wichtigkeit	CVSS v3 Rating
Kritisch	9.0 – 10.0
Hoch	7.0 – 8.9
Medium	4.0 – 6.9
Low	0.1 – 3.9

CVSS bietet eine numerische (0-10) Darstellung des Schweregrads einer Sicherheitslücke in der Informationstechnologie. Je höher der Wert desto schlimmer ist das Schadensausmass beim Eintreten des Risikos. CVSS-Scores werden üblicherweise von Sicherheitsteams als Teil eines Schwachstellen-Management-Programms verwendet, um einen Vergleichspunkt zwischen Schwachstellen zu schaffen und um die Prioritäten bei der Behebung von Schwachstellen festzulegen. Der CVSS Score setzt sich aus den Messungen jeder der folgenden Metriken zusammen:

- Angriffsvektor (AV)
- Angriffskomplexität (AC)
- Erforderliche Privilegien (PR)
- Benutzer-Interaktion (UI)
- Umfang (S)
- Vertraulichkeit (C)
- Integrität (I)
- Verfügbarkeit (A)

Man kann den CVSS Score für eigene Schwachstellen mit online verfügbaren CVSS Rechnern berechnen lassen.

Entscheid

Da sich diese Arbeit auf das Testen von Schwachstellen fokussiert wurde entschieden, die Schwachstellen nach ihrer Testbarkeit zu priorisieren und nicht wie üblich nach dem CVSS Score.

Methodik

Für die Priorisierung der Schwachstellen wurde das nachfolgende Template erstellt und verwendet:

Schwachstelle	OWASP Schwachstelle
STRIDE Type	Der jeweilige STRIDE Type z.B Spoofing
Angriffe	Möglicher Angriff der die Schwachstelle ausnützt
Description	Beschreibung der Schwachstelle
Mitigation Massnahmen	Sicherheitsmassnahmen, die implementiert werden können, um das Risiko zu mindern
Testbarkeit	Wie kann die Schwachstelle oder die eingesetzte Massnahme getestet beziehungsweise überprüft werden
Priorität	Welche Priorität hat die Schwachstelle
Begründung	Begründung für die gewählte Priorität

1.7.4.1 Threats Keycloak Authentification Service

Einleitung

Keyloak ist ein OAuth2 Authentfizierungsserver wie viele andere und hat seine ganz eigenen Schwachstellen, welche in dieser Arbeit nicht tiefer betrachtet werden. Eine gute Liste von Schwachstellen und Massnahmen, um sie zu entschärfen, sind im RFC 6819 OAuth 2.0 Threat Model and Security Considerations [38], welcher von der IETF veröffentlicht wurde zu finden.

Bedrohungen

Die Keycloak Admin Console ist eine JavaScript/HTML5 Webapplikation, die REST-Aufrufe an die Keycloak Admin API weiterleitet. Der Keylcoak Service muss deshalb wie eine Webapplikation betrachtet werden. Die üblichen Schwachstellen einer Webapplikation sind auch hier zu beachten. Da dies jedoch den Rahmen der Arbeit übersteigt, wurde mit dem Praxispartner entschieden diese in der Arbeit nicht weiter zu thematisieren. Da die Authentifizierung der API auf den Keycloak Service ausgelagert wurde, hat sich jedoch auch die nachfolgende API-Schwachstelle bezüglich der Authentifizierung auf diesen Service verschoben:

Schwachstelle	**Broken Authentification (OWASP API Nr. 2)**
STRIDE Type	Spoofing
Angriffe	Brut-Force, Credential-Stuffing, Man in the Middle, XSS
Description	Ein nicht korrekt implementierter Authentifizierung-Mechanismus ist gemäss OWASP (vgl. [39]) die zweithäufigste Schwachstelle einer API. Alle Schwachstellen, die mit Authentifizierung zu tun haben, hängen grundsätzlich mit einer schlechten Verwaltung der Passwörter oder einem gebrochenen Anmelde-Mechanismus zusammen. Die Angreifer kompromittieren die Authentifizierung, um Zugriff auf die Daten der jeweiligen Benutzer zu erhalten. Beispielsweise wenn es dem Angreifer möglich ist mittels „Credential Stuffing“ [40] eine beliebige Anzahl von Versuchen durchzuführen, um Zugang zu erhalten. Die Schwachstelle hängt auch mit den verschiedenen Ansätzen zur Authentifizierung zusammen. Es gibt verschiedene Mechanismen wie beispielweise Basic Authentification oder Single Sign-On (SSO) Jeder hat individuelle Schwachstellen, für welche die OWASP Community eigene Dokumentationen bereitstellt. Der grosse Unterschied zu einer Webapplikation ist, dass bei APIs ein API-Key / Access Token anstelle einer Session verwendet wird. Weil der Server die Session nicht mehr verwaltet, muss der Client sich bei jeder Anfrage authentifizieren, indem er den API-Key mitsendet.
Mitigation Massnahmen	• Strenge Passwort Regeln (Password Policies) • Anmeldeverfahren darf nicht umgangen werden • Kurzlebige Zugriffstoken verwenden • Verwende strengere Ratenbegrenzung für die Authentifizierung • Brute-Force Angriffe auf einen Account blockieren, Implementieren von Sperrrichtlinien • Keycloak muss SSL/HTTPS verwende, was standardmässig nicht der aktiviert ist.

Testbarkeit	Es kann manuell mit Postman getestet werden, ob der Session Token nach jedem Anmeldevorgang neu zugewiesen wird. Falls der Token ausläuft, kann geprüft werden, ob der Zugriff entsprechend blockiert und ein HTTP Error 401 (Unauthorized) zurückgeben wird. APIs kommunizieren neben den Clients häufig auch mit anderen APIs. Normalerweise wird der Session-Token des Clients für die weitere Kommunikation verwendet. Entwickler halten sich jedoch nicht immer an diese Praxis. Sie verwenden oft spezielle Tokens für die Kommunikation von API zu API. Dadurch hat die API möglicherweise Zugriff auf mehr Daten als beabsichtigt. Eine solche Schwachstelle kann nur in einem Code- oder Architektur-Review gefunden werden (vgl. [41]). Es kann getestet werden, ob der Zugriff mit einer falschen Audience im JWT Access Token abgelehnt wird.
Priorität	2
Begründung	Da die Schwachstelle auf den Authentifizierungs-Service ausgelagert wurde, ist sie mit Priorität 2 bewertet worden und kann in der Zukunft geplant oder bei genügend Zeitreserven noch behandelt.

1.7.4.2 Threats API-Gateway Tyk

Einleitung

Der API-Gateway fungiert als Hauptpunkt für den API-Verkehr. Durch ihn geht sämtlicher Verkehr, der an die dahinterliegenden Microservices weitergeleitet werden soll. Bei dieser Arbeit soll der API Gateway die nachfolgenden zwei Aufgaben übernehmen:

- Ressource & Rate Limiting
- Logging & Monitoring
- Versionierung

Der Tyk Gateway ist ebenfalls wie auch der Keycloak-Service eine Web-Applikation. Tyk hat deshalb selbst eine Liste von Schwachstellen und Massnahmen, um sie zu entschärfen, publiziert (vgl. [42]).

Bedrohungen

Durch die definierten Aufgaben des API-Gateways haben sich die nachfolgenden Schwachstellen der OWASP Top 10 Liste für APIs massgeblich durch den API Gateway sicherzustellen:

Schwachstelle	**Lack of Resources & Rate Limiting (OWASP Nr. 4)**
STRIDE Type	Denial of Service
Angriffe	DOS / DDOS
Description	Es ist wichtig, Einschränkungen für die Grösse und/oder Anzahl der Ressourcen festzulegen, die ein Client über eine API anfordern kann. Wenn keine Ratenbegrenzung implementiert ist, können Cyberkriminelle API-Server von der Leistungsseite her missbrauchen, was zu einem Denial of Service (DoS) oder Brute Force führt. Angreifer können APIs gezielt angreifen, um Prozeduren aufzurufen, welche viel Memory verbrauchen und so den Server lahmlegen (vgl. [39]). Ein nicht vorhandes Rate Limiting ergibt dem Angreifer überhaupt die Möglichkeit einen Brut-Force Attack durchzuführen. So konnten im Jahr 2020 erfolgreich Passwörter vom Videokonferenz-Programm „Zoom“ erraten und gestohlen werden (vgl. [43])
Mitigation Massnahmen	• Rate Limiting im API Gateway einsetzten Weiter mögliche Massnahmen, welche in der Arbeit aber nicht weiter betrachtet werden könnten sein: • IP Blocking einsetzten • Payload Size Limits • Check Compression Ratios wenn Zip angeboten wird • Wenn man XML, YAML oder andere Konfigurationsfiles anbietet muss man Parser Checks um Rekursionen zu verhindern • Der Applikationsserver auf welchem das Backend läuft müssen Limiten auf die CPU Auslastung und Memory Usage gesetzt werden (Docker / Kubernetes Config)
Testbarkeit	Es kann automatisiert getestet werden, ob ein automatischer Angriff wie Brut-Force nach einer entsprechenden Anzahl an

	Versuchen, verhindert wird. Eine Strategie ist es festzulegen, dass für einen API-Endpoints in einem bestimmten Zeitrahmen nur eine festgelegte Anzahl an Request zulässt. Clients, welche das Limit aufgebraucht haben, sollten keinen Daten mehr erhalten und stattdessen den Statuscode 429 (Too Many Requests) erhalten. (vgl. [41]) Es kann somit zuerst manuell getestet werden, ob die entsprechenden Rate Limits korrekt funktionieren. Diese manuellen Tests können dann später mittels CI/CD automatisiert werden.
Priorität	2
Begründung	Da die Schwachstelle auf den API-Gateway-Service ausgelagert wurde, ist sie mit Priorität 2 bewertet worden und kann in der Zukunft geplant oder bei genügend Zeitreserven noch behandelt.

Schwachstelle	**Insufficient Logging & Monitoring (OWASP Nr. 10)**
STRIDE Type	Reputation
Angriffe	Trifft für alle Angriffe zu
Description	Die letzte Schwachstelle bezeichnet die unzureichenden Protokollierungs- und Überwachungsverfahren. Man muss sicherstellen, dass alles was in einer API passiert, nachverfolgt werden kann. Es sollten immer Protokolle vorhanden sein, die genau zeigen, was der Angreifer für Versuche unternommen hat. (vgl. [39])
Mitigation Massnahmen	Netzwerk, Applikations- und Datenbank-Logging muss betrieben werden.
Testbarkeit	Kann nicht getestet werden. Nur durch manuelle Überprüfung beispielsweise mit Code Reviews überprüfbar.
Priorität	3
Begründung	Es wurde entschieden, dass Schwachstelle aufgrund der nicht verfügbaren Testbarkeit in dieser Arbeit nicht weiter vertieft behandelt wird.

1.7.4.3 Threats Trading API

Einleitung

Die Trading API, welche mit Spring Boot entwickelt wurde

Für den API Gateway Tyk wurden die nachfolgenden STRIDE-Bedrohungen identifiziert:

Schwachstelle	**Broken Object Level Autorisation (OWASP Nr 1)**
STRIDE Type	Information Disclosure
Angriff	Parameter Manipulation
Description	Angreifer nutzen API-Endpunkte aus, welche eine gebrochene Autorisierung zulassen. Es ist der häufigste und folgenreichste Angriff auf APIs. Indem Sie die auf nicht veröffentlichte Endpunkte zugreifen oder die ID eines Objekts manipulieren, welches bei einer Anfrage gesendet wird. Dies führt zu einem unberechtigten Zugriff auf sensible Daten. Gemäss OWASP (vgl. [39]) vergessen viele Entwickler, die Prüfungen vor dem Zugriff auf ein sensibles Objekt durchzuführen. Nehmen wir an, ein Benutzer erzeugt ein Dokument mit der ID 1. Ihm sollte nur der Zugriff auf dieses Dokument erlaubt sein. Bei einer anderen ID z.B. 2, sollte der Aufruf einen 403 (Forbidden) Error zurückgeben. Oft werden nicht alle Endpoints auf Zugriffskontrollen geprüft.
Mitigation Massnahmen	• Verwendung von Open ID Connect Standard • Implementieren von Authorization Checks mit User Policies • Verwendung von zufälligen IDs (UUID) • Es wird die ID, welcher in der Session gespeichert ist verwendet und nicht auf die vom Client gesendete ID vertraut
Testbarkeit	Diese Schwachstelle kann nicht automatisiert getestet werden, da es dafür kein Tool gibt, welches die Schwachstelle automatisch erkennen kann. Die Schwachstelle muss manuell für jede API getestet werden. Um die Schwachstelle zu testen, wird eine andere ID in der URL oder als Teil eines Query-Parameters übergebe, um zu prüfen, ob die Response Daten beinhaltet oder nicht. Die manuell erstellten Tests können später als Teil von Integration Tests automatisiert werden (vgl. [41]). Die automatische Erkennung der Schwachstelle mittels eines DAST-Tools ist aktuell nicht möglich.
Priorität	1
Begründung	Da die Schwachstelle eine starke Verletzung der Vertraulichkeit darstellt, wird sie bei der Arbeit vertieft behandelt und auch getestet. Es ist gemäss OWASP Top 10 die häufigste Schwachstelle und sie kann mit manuellen Postman Tests erkannt und verhindert werden.

Schwachstelle	Excessive Data Exposure (OWASP Nr 3)
STRIDE Type	Information Disclosure
Angriff	Parameter Manipulation oder gar kein Angriff notwendig, wenn beispielsweise ganzes Objekt zurückgegeben wird und nur durch SPA gefiltert.
Description	Diese Schwachstelle ist oft eine Ursache von schlechter Kommunikation. Oftmals werden Frontend- und Backend-Entwicklung voneinander getrennt und von verschiedenen Teams implementiert. Der Frontend Developer bittet den Backend-Developer im Projekt dann beispielsweise um Daten über einen User. Der Backend Developer bietet ihm dann eine Route an mit welcher er für eine bestimmte User-ID das ganze User Objekt zurückerhalt. Der Frontend Developer benötigt aber meistens aber nur Teile davon wie z.B den Namen und die Adresse. Zwei Wochen später möchte er vielleicht noch die Telefonnummer im Frontend anzeigen, was wiederum zu einer Änderung in der API führt. Es ist für den API-Entwickler einfacher und bequemer alle Informationen zu einem Objekt über einen Endpoint anzubieten, weshalb diese Schwachstelle in der Realität öfter auftritt als man annehmen würde (vgl. [39])
Mitigation Massnahmen	• Erzwingen von Antwortprüfungen, um versehentliche Datenlecks zu verhindern • Die Daten müssen von der API entsprechend gefiltert werden und nicht vom Client. • Definieren von Schemas für alle API-Antworten • Identifizieren von sensiblen Informationen und die Verwendung begründen
Testbarkeit	Die Schwachstelle kann nicht automatisch getestet werden, da kein Tool wissen kann, welche Antworten zu viel Informationen beinhaltet. Für jede API, muss analysiert werden welche Informationen sie zurückgibt, um zu prüfen, ob sie mehr Informationen als nötig exponiert. (vgl. [41]) Mittels manuellen Tests kann man das Schema einer Antwortnachricht überprüfen. Diese können dann später automatisiert werden, was dazu führt das bei einer Änderung des Schemas der Test fehlschlägt und so den Entwickler bewusst darauf hinweist das nun andere Daten exponiert werden. Die Schwachstelle kann dennoch auftreten, wenn beispielsweise eine zu grosse Liste an den Benutzer zurückgegeben wird.
Priorität	1
Entscheid	Es sollen Schema Tests für die entsprechenden Response Nachrichten erstellt werden, um bei einer Änderung der Response-Nachricht auch eine Änderung des Tests zu erzwingen. Damit wird verhindert das unwissentlich die Response-Nachricht verändert wird.

Schwachstelle	Broken Function Level Authorization (OWASP Nr 5)
STRIDE Type	Elevation of Privilege
Angriff	Parameter Manipulation oder gar kein Angriff notwendig da Fehler in Zugriffssteuerung
Description	Autorisierungsfehler können auftreten, wenn es komplexe Richtlinien zur Zugriffskontrolle gibt. Wenn Funktionsebenen nicht konsolidiert und vereinfacht werden, können Angreifer Zugriff auf Ressourcen und Verwaltungsfunktionen erhalten. Wenn z.B. ein normaler Benutzer versucht ein Administrator zu werden oder anstelle eines Get Requests einen Delete Request sendet. (vgl. [39])
Mitigation Massnahmen	• Nicht auf Informationen vom Client verlassen • Lehne jeden Zugriff standardmässig ab • Erlaube Operationen nur User, welche die entsprechende Rolle haben
Testbarkeit	Es kann manuell getestet werden, ob ein beispielsweise ein DELETE Request mit einem normalen Useraccount den Statuscode 403 (Forbidden) als Antwort erhält. Wenn die API zweihunderter Codes wie 200 (OK), 204(NO Content) oder 201 (CREATED) zurückgibt ist die Schwachstelle vorhanden. (vgl. [41]).
Priorität	1
Entscheid	Wenn ein Entwickler bei der Entwicklung der API nicht alle Zugriffspunkt korrekt absichert, ist so ein Fehler schnell im System. Eine solche Schwachstelle kann mit einer klaren Definition der Zugriffsrechte effizient mit Postman getestet werden. Es wurde deshalb mit dem Praxispartner entschieden diese Schwachstelle vertieft zu behandeln und zu testen.

Schwachstelle	Mass Assigment (OWASP Nr 6)
STRIDE Type	Elevation of Privilege
Angriff	Parameter Manipulation
Description	Man möchte irgendwelche Daten ändern beispielsweisse die Adresse seines Accounts. Es wird ein Post mit einem JSON geschickt, welches die neuen Werte enthält. Model Programming Languages wie Node JS und Ruby nehmen ein JSON und schreiben es direkt in die Datenbank. Wenn nun ein Angreifer das JSON um neue zusätzliche Properties erweitert, werden diese ohne Schutzmassnahmen in die Datenbank gespeichert. Wenn nun die Datenbank auf Properties wie isAdmin: true entsprechend

	reagiert hat man die die Schwachstelle erfolgreich ausgenutzt (vgl. [39])
Mitigation Massnahmen	• Kein automatisches Binden von einkommenden Daten zu einem Objekt. • Explizit alle zu erwarteten Parameters und Payload definieren. • Präzise Definition in der Design Phase von Schemas, welche in einem Request akzeptiert werden • Korrekte Inputvalidierung
Testbarkeit	Beim Testen muss geprüft werden, dass jeder Parameter separat zugewiesen wird. Anders gesagt, wenn man nur Name und Passwort annehmen will, muss dies explizit definiert werden. Es kann manuell getestet werden ob zusätzliche Parameter in einem Request entsprechend abgelehnt werden (vgl. [41]).
Priorität	1
Entscheid	Es sollen Validierungstests der einzelnen Attribute für die entsprechenden Response Nachrichten erstellt werden, um zu prüfen ob durch einen modifizierten Request neue Attribute gesetzt oder verändert werden können.

Schwachstelle	**Security Misconfiguration (OWASP Nr 7)**
STRIDE Type	Information Disclosure / Elevation of Privilege
Angriffe	Man in the Middle , Parameter Manipulation
Mitigation Massnahmen	• Immer TLS verwenden • Unnötige HTTP Methoden wie beispielsweise HEAD nicht zulassen • Keine technischen Details in Error-Meldungen an den Client senden • Falls CORS verwendet wird müssen die Header korrekt gesetzt werden. Auf keinen Fall * verwenden
Testbarkeit	Es kann getestet werden, ob die Applikation keine Internal Server Error Meldungen (Statuscode 500) zurückgibt. Diese beinhalten oft technische Details der Applikation. Weiter kann geprüft werden, ob Security Header wie HSTS (HTTP Strict Transport Security) oder CSP vorhanden ist. Sobald eine Webapplikation die API verwendet, kann auf CORS Header getestet werden.
Priorität	1
Entscheid	Es wurde entschieden das eine CSP eingesetzt wird und diese überprüft werden soll.

Schwachstelle	Injection (OWASP Nr 8)
STRIDE Type	**Information Disclosure / Elevation of Privilege**
Angriff	**Parameter Manipulation mit Injection.**
Description	Oftmals injizieren Angreifer bösartigen Code als Teil eines Befehls oder einer Abfrage. Mit SQL, NoSQL oder Kommando-Befehlen können Angreifer ein System weiter kompromittieren und schweren Schaden anrichten. In der OWASP-Top-10-Liste der Sicherheitsrisiken für Webanwendungen nehmen Injections den ersten Platz ein. Injections für APIs belegen jedoch nur den achten Platz. Das liegt gemäss OWASP daran, dass moderne Frameworks und Patterns uns vor den primitivsten SQL und XSS Angriffen bewahren.
Mitigation Massnahmen	• Kein automatisches Binden von einkommenden Daten zu einem Objekt. • Explizit alle zu erwarteten Parameters und Payload definieren. Präzise Definition in der Design Phase von Schemas, welche in einem Request akzeptiert werden
Testbarkeit	Injections können mit Tools wie ZAP oder Burp Suite automatisiert getestet werden (vgl. [41]). Es können auch manuelle Injection Angriffe mit Postman durchgeführt werden, um einen bestimmten Endpoint zu Testen. Häufig setzt sich ein Fehler eines Entwicklers in einem Endpoint auf die anderen Endpoints fort. Dadurch kommt es vor das eine SQL-Schwachstelle auf mehrere Ressourcen zutrifft.
Priorität	1
Entscheid	Es wurde mit dem Praxispartner festgelegt Injections zu testen, da sie automatisiert mit einem DAST-Tool getestet werden können. Es soll bewusst ein Endpoint in der Spring API entwickelt, der eine SQL-Schwachstelle darstellt, um zu überprüfen, ob sie durch ein SAST oder DAST Tool automatisch erkannt wird.

Schwachstelle	Improper Asset Management (OWASP Nr. 9)
STRIDE Type	**Elevation of Privilege**
Angriff	**Kein bestimmter Angriff vorhanden**
Description	Alte API-Versionen sind in der Regel nicht gepatcht und für einen Angreifer besonders attraktiv, um einen Angriff durchzuführen. Die alten Versionen haben nicht die aktuellen Sicherheitsmechanismen implementiert, die die neuste API-Version entsprechend schützen. Veraltete Dokumentation erschweren das Auffinden von potenziellen Schwachstellen. Fehlende Strategien für die führen häufig zum Betrieb von alten Versionen. Es ist üblich, veraltete und unnötige APIs zu finden. Dies hat damit zu tun, dass das Bereitstellen einer Umgebung heute ganz einfach und unabhängig, durch moderne Konzepte wie die Microservice-Architektur und Cloud-Computing erfolgt (vgl. [39])
Mitigation Massnahmen	• Die API-Dokumentation darf nur berechtigten Personen zur Verfügung stehen • Dokumentation automatisch erstellen mittels CI/CD damit dieser immer aktuell sind
Testbarkeit	Kann nicht getestet werden, da eine solche Schwachstelle immer sehr individuell sind und von der jeweiligen API Version abhängig sind. Es kann in Architektur-Reviews überprüft werden ob die Dokumentationen korrekt und ausführlich sind.
Priorität	3
Entscheid	Es wurde entschieden, dass Schwachstelle aufgrund der nicht verfügbaren Testbarkeit in dieser Arbeit nicht weiter vertieft behandelt wird.

1.7.4.4 Resultate

Resultate

Die nachfolgenden Schwachstellen wurden mit Priorität 1 bewertet und werden deshalb getestet und vertieft betrachtet:

- Broken Object Level Autorisation (OWASP Nr 1)
- Excessive Data Exposure (OWASP Nr 3)
- Broken Function Level Authorization (OWASP Nr 5)
- Mass Assigment (OWASP Nr 6)
- Security Misconfiguration (OWASP Nr 7)
- Injection (OWASP Nr 8)

Die Schwachstellen Broken Authentification (OWASP API Nr. 2) und Lack of Resources & Rate Limiting (OWASP Nr. 4) sind testbar aber befinden sich auf anderen Komponenten, weshalb erst umgesetzt werden, wenn alle Schwachstellen mit Priorität 1 getestet worden sind. Die Schwachstelle Improper Asset Management (OWASP Nr. 9) und Insufficient Logging & Monitoring (OWASP Nr. 10) können nicht durch ein Tool überprüft werden und sind deshalb nicht Bestandteil dieser Arbeit.

2 Software-Engineering

Inhalt Im folgenden Kapitel wird die Arbeit aus Software-Engineering Sicht betrachtet und es werden die nachfolgenden Aspekte der Softwareentwicklung beschrieben:

- Funktionale und nichtfunktionale Anforderungen
- Domainanalyse
- Systemübersicht
- Logische Architektur

2.1 Anforderungsspezifikation

2.1.1 Funktionale Anforderungen

Inhalt Im folgenden Kapitel werden alle funktionalen Anforderungen definiert und mit Hilfe von Use-Cases dargestellt.

2.1.1.1 Aktoren

Customer Ein angemeldeter Benutzer, welcher die API benutzt um seine Handelspositionen anzuschauen und neue Transaktionen erstellen kann.

Administrator Ein Benutzer, der bereits angemeldet ist und die Rechte besitzt, um die Kosten für einen Auftrag anzupassen. Er kann die Kosten eines Auftrags verringern oder erhöhen. Er hat jedoch keine Ansicht auf die einzelnen Handelspositionen von Kunden

Not authenticated User Ein nicht angemeldeter Benutzer, welcher keinen Zugriff erhält, bis er sich bei einem vertrauenswürdigen Identity Provider angemeldet hat

2.1.1.2 Use Cases Übersicht

Diagramm In der folgenden Abbildung sind die Use Cases, welche sich auf die API beziehen, ersichtlich. Der Fokus dieser Bachelorarbeit liegt auf der Umsetzung einer API, welche Security Tests implementiert. Das Augenmerk wird daher auf die entsprechenden Anwendungsfällegelegt.
Die einzelnen Use Cases werden in einem weiteren Schritt detaillierter beschrieben und priorisiert.

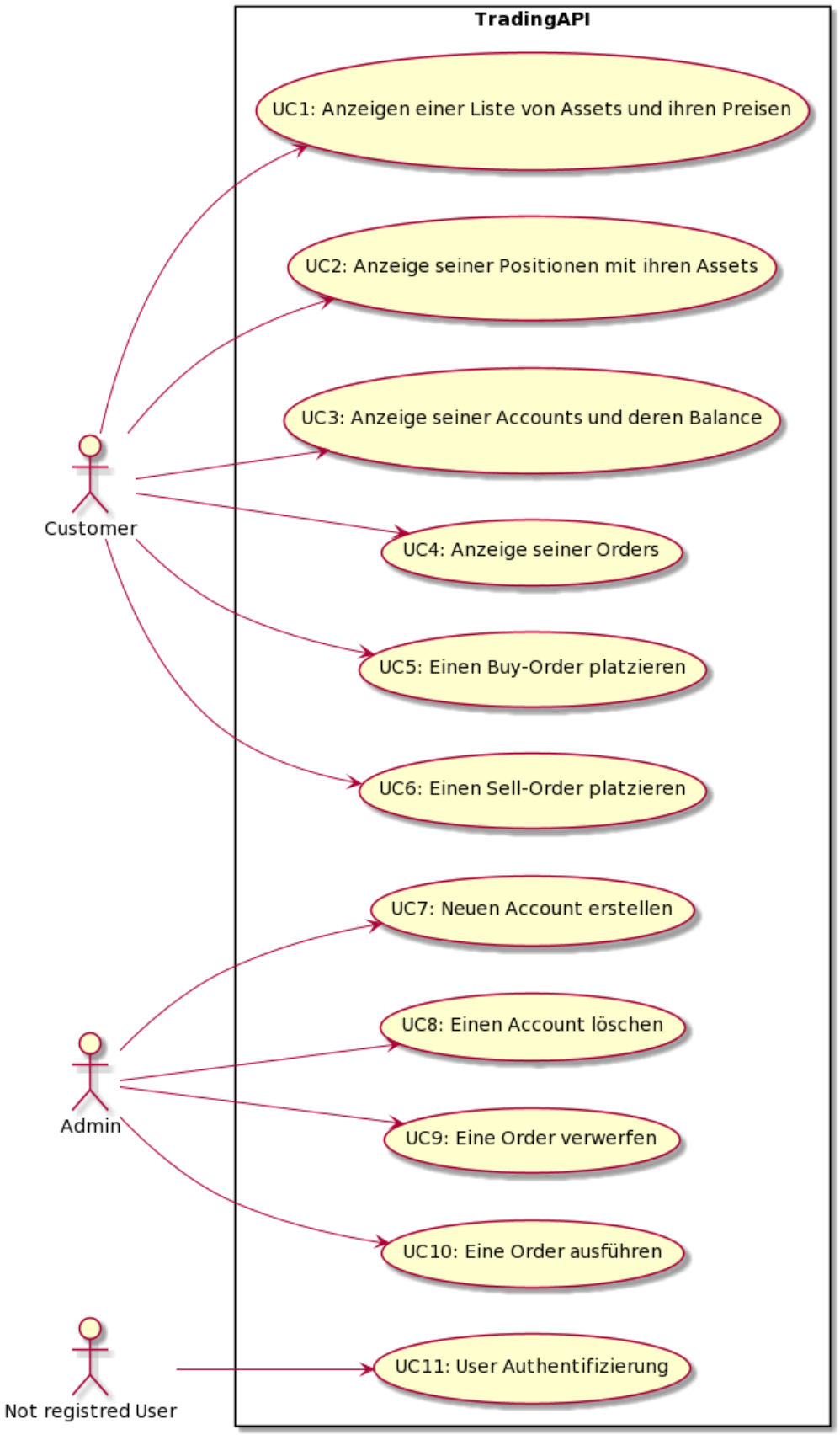


Abbildung r Trading API Use Case Diagramm

2.1.1.3 Use Cases Beschreibung

UC01: Anzeigen einer Liste von Assets	**Aktor:** Customer Einem Kunden wird eine Liste von Assets (Wertschriften) angezeigt, für welche er berechtigt ist einen Buy-Order zu platzieren.
UC02: Anzeige seiner Positionen	**Aktor:** Customer Dem Kunden wird eine Liste seiner Positionen für Assets, welche er besitzt, angezeigt. Die Position zeigt dem Kunden den aktuellen Wert der Position und die Anzahl der vorhandenen Assets.
UC03: Anzeige seiner Accounts	**Aktor:** Customer Dem Kunden wird eine Liste seiner Accounts angezeigt. Ein Account beinhaltet alle Positionen, welche ein Kunde besitzt. Zusätzlich sind auf dem Account einige personenbezogen Daten enthalten wie Vorname, Nachname und Adresse.
UC4: Anzeige seiner Orders	**Aktor:** Customer Dem Kunden wird eine Liste aller seiner Orders angezeigt. Eine Order gehört immer zu einer bestimmten Position. Demzufolge soll er auch die Möglichkeit haben alle Orders einer Position anzusehen.
UC05: Einen Buy Order platzieren	**Aktor:** Customer Der Kunde platziert eine Kaufauftrag für ein bestimmtes Asset, welches er kaufen möchte. Er kann eine bestimmte Anzahl wählen und einen Ausführungspreis festlegen.
UC06: Einen Sell Order platzieren	**Aktor:** Customer Der Kunde platziert eine Verkaufsauftrag für ein bestimmtes Asset, welches er kaufen möchte. Er kann eine bestimmte Anzahl wählen und einen Ausführungspreis festlegen.
UC07: Neuen Account erstellen	**Aktor:** Admin Ein Administrator kann neue Accounts für einen Kunden anlegen. Erst wenn ein Kunde einen Account besitzt, kann er Positionen besitzen und Orders erstellen.
UC08: Account löschen	**Aktor: Admin** Ein Administrator kann einen Account für einen bestimmten Benutzer löschen.
UC09: Eine Order verwerfen	**Aktor:** Admin Ein Administrator kann eine platzierte Order verwerfen, indem er den Status des Orders auf „Discarded“ stellt
UC10: Eine Order ausführen	**Aktor:** Admin

	Ein Administrator kann eine platzierte Order ausführen, indem er den Status des Auftrags auf „DONE“ stellt.
UC11: Authentifizierung	**Aktor:** Not registered User Ein nicht angemeldeter Benutzer muss sich an der API anmelden können, um auf bestimmte Ressourcen berechtigt zu werden.

2.1.2 Nichtfunktionale Anforderungen

Methodik Für die nichtfunktionalen Anforderungen werden die Qualitätsmerkmale von FURPS [44] verwendet. Für eine zielführende Definition der nichtfunktionalen Anforderungen wird der SMART [45] Ansatz gewählt.

2.1.2.1 Funktionalität

Richtigkeit Endpoints sollen nur Daten zurückliefern, welche auch wirklich benötigt werden. Es wird sichergestellt das kein schädlicher Code in das Backend eingeschleust werden kann.

Sicherheit Alle Endpoints sollen entsprechend abgesichert werden damit kein unberechtigter Zugriff erfolgen kann. Die Authentifizierung soll mit Open ID Connect und die Autorisierung mit dem Protokoll OAuth2 erfolgen.

2.1.2.2 Benutzbarkeit

Verständlichkeit Die eindeutigen Endpoints werden mit Open API dokumentiert. Das Nachrichtenformat der Request und Response Meldungen wird dort ebenfalls dokumentiert, damit dem Benutzer klar ist, wie er die unterschiedlichen Endpoints verwenden kann.

2.1.2.3 Zuverlässigkeit

Reife Die Funktionen der API soll in 90% der Fälle wie geplant funktionieren. Systemfehler werden benutzter nicht angezeigt. Ein Konzept für das Logging wurde noch nicht erarbeitet. Ein Konzept für ein Logging von Systemfehlern könnte für die Zukunft geplant werden.

Fehlertoleranz Die Request Meldungen an Endpoints dürfen nicht vom Schema abweichen und müssen bei Nichteinhaltung eine entsprechende Fehlermeldung generieren. Mindestens 90% der Endpoints müssen unzulässige Eingaben abfangen.

Wiederherstellbarkeit Gelöschte Daten können nicht wiederhergestellt werden. Die Sicherung der Daten ist nicht Teil dieser Arbeit und kann für die Zukunft geplant werden. ➔ Keine Anforderung, da bei Restart Testumgebung neu erstellt wird

2.1.2.4 Effizienz

Zeit für Responses Anfragen an den API Server sollen in maximal 5 Sekunden erfolgen.

2.1.2.5 Wartbarkeit

Modifizierbarkeit Durch die Verwendung diverser Patterns wurde auf eine Abstraktion der Software gesetzt. Die verschiedenen Schichten der Backend-Applikationen lassen sich dank Dependency Injection einfach austauschen.

Prüfbarkeit Jede Code Änderung wird versioniert auf dem Gitlab Server der Ost gespeichert.

Testbarkeit Bei einer Code Anpassung wird mittels CI/CD immer geprüft, wie hoch die Test Coverage ist. Beim Builden des Projekts werden automatisch Unit- und Integratio-Tests ausgeführt. Wenn einer der Tests fehlschlägt wird der Build abgebrochen.

2.1.2.6 Übertragbarkeit

Installierbarkeit / Anpassbarkeit Die API ist mit Spring Boot umgesetzt und es wird ein Dockerfile erstellt. Damit kann die API auf allen Plattformen installiert werden. Die Abstraktation der Datenbank erlaubt deren Austausch

.

2.2 Architektur

Inhalt In diesem Kapitel wird die Architektur der zu entwickelnden API beschrieben. Architekturentscheide werden begründet und verwendete Technologien aufgezeigt.

2.2.1 Domainanalyse

Domainmodel Nachfolgendes Bild zeigt das Domainmodell der Applikation.

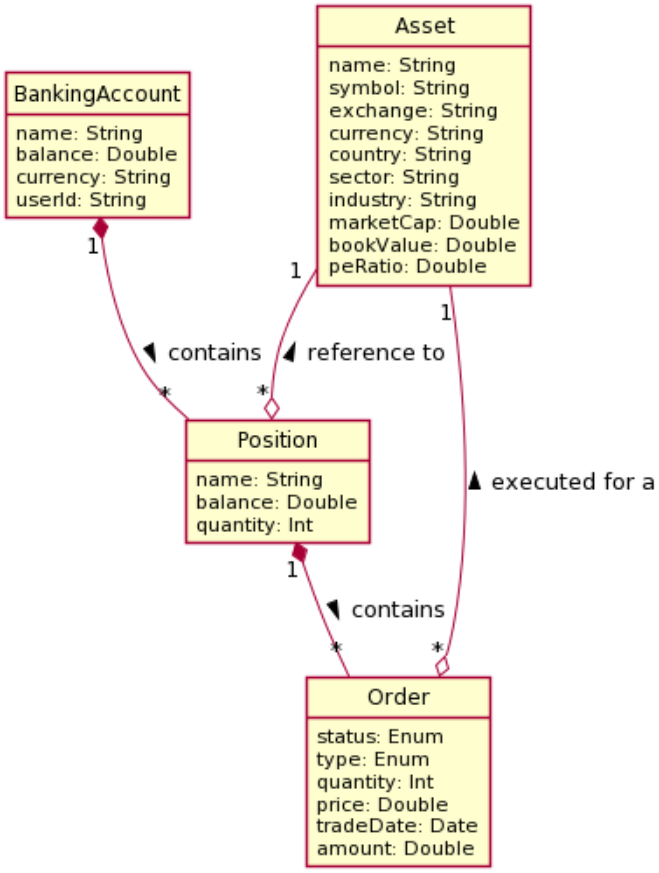


Abbildung 18: Trading API Domainmodel

Asset Ein Asset ist ein börsengehandeltes Wertpapier, welches ein Kunde an einer Börse kaufen oder verkaufen kann. Asset-Informationen sind für jeden frei zugänglich.

Banking Account Ein Banking Account wird durch einen Administrator für einen Kunden erzeugt. Ein Banking Account ist pro Benutzer und Währung einzigartig. Er enthält alle Positionen und Orders in der gleichen Währung. Ein Banking Account hat einen Namen und eine Balance.

Position Eine Position wird automatisch erstellt, wenn der Benutzer das erste Mal ein bestimmtes Wertpapier kauft. Alle nachfolgenden Kauf- und Verkaufsaufträge werden in der Position gesammelt. Die Balance einer Position zeigt den aktuellen Wert der Position und die Quantity die vorhandene Anzahl der Wertschriften.

Order Ein Benutzer der API kann eine Order erstellen, um Wertschriften zu kaufen oder zu verkaufen. Er die Menge und einen gewünschten Preis. Wenn sich ein Käufer oder Verkäufer findet, wird die Order durch den Administrator ausgeführt. Ein Administrator kann auch eine noch nicht ausgeführte Order für einen Kunden stornieren. Der Kunde selbst kann dies nicht.

2.2.2 Systemübersicht

Inhalt Nachfolgendes Bild zeigt das Deployment Diagramm aller eingesetzten Komponenten und die Protokolle, welche bei der Arbeit eingesetzt werden. Die API wurde auf der Heroku Cloud Plattform bereitgestellt und alle Daten werden in einer Postgres Datenbank in der Amazon AWS Cloud gespeichert. Der Authentifizierungs-Service wurde mit der Keycloak Software umgesetzt und ebenfalls auf Heroku bereitgestellt. Der Keycloak-Service benötigt ebenfalls eine Postgres Datenbank, welche in der AWS Cloud aufgesetzt wurde.

Der API Gateway von Tyk besteht aus zwei einzelnen Komponenten, dem Gateway selbst und einem Dashboard, welches eine grafische Oberfläche für die Einstellungen am Gateway liefert. Für das Dashboard musste eine MongoDB Datenbank auf Atlas erstellt werden.

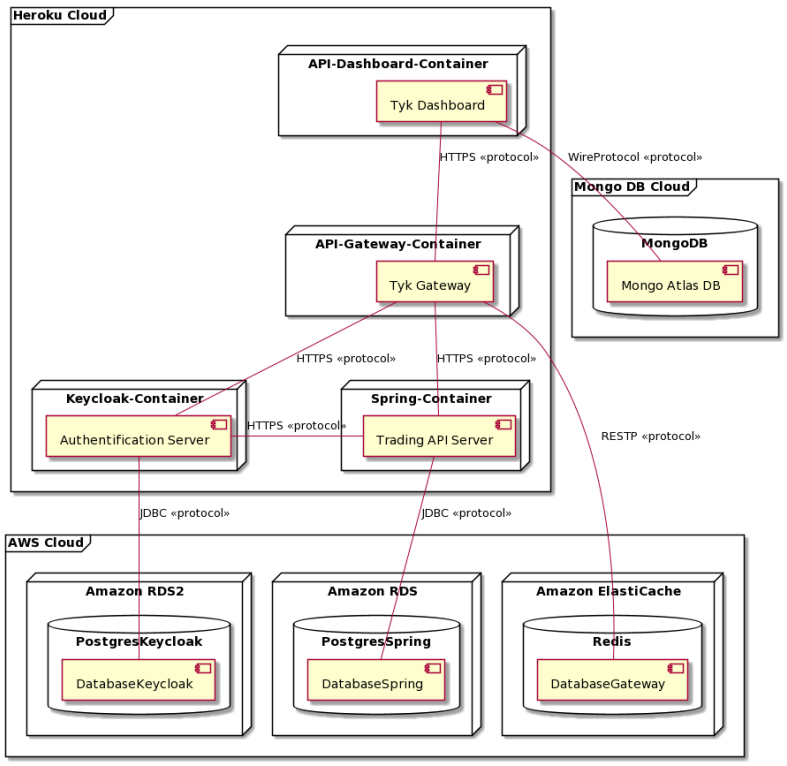


Abbildung 19: Systemübersicht Alle Komponenten

Entscheid Heroku

Die verwendeten Komponenten werden auf Heroku gehosted. Heroku bietet eine schnelle, zuverlässige und skalierbare Lösung um eine Applikation im Web bereitzustellen. Damit während der Arbeit keine Zeit mit durch die Verwaltung von Servern benötigt wird wurde entschieden ein Platform-as-a-Service-Solution (PaaS) zu verwenden. PaaS-Architekturen umfassen in der Regel Betriebssysteme, Ausführungsumgebungen für Programmiersprachen, Bibliotheken, Datenbanken, Webserver und Konnektivität zu einigen Plattformen.

Heroku, als eine PaaS Lösung ist generell einfach zu bedienen Heroku hat ein kostenloses Servicemodell für kleine Projekte. Mit Heroku können erstellte Applikationen in der Cloud-Infrastruktur sehr einfach bereitgestellt werden. Man hat nur begrenzten Zugriff auf die Konfiguration der Hosting-Umgebung da diese Ressourcen vom PaaS Anbieter bereitgestellt werden. Man verwaltet oder kontrollieren nicht die zugrunde liegende Cloud-Infrastruktur einschliesslich Netzwerk, Server, Betriebssysteme oder Speicher. Dadurch kann man sich auf die Entwicklung und Konfiguration der Applikationen fokussieren.

2.2.3 Logische Architektur

Schichtenmodell

Für die logische Architektur der Applikation wurde das «Model View Controller»-Architekturmuster entschieden. Zwischen Controller und Model Layer wurde noch ein Service Layer hinzugefügt. Das MVC Muster wurde gewählt, da es für die Entwicklung einer Spring Boot Applikation eine weithin akzeptierte Lösung ist, um die Codebasis in drei eigene Schichten mit unterschiedlichen Verantwortlichkeiten zu unterteilen. Dieses Muster erleichtert es, die Applikation zukünftig einfach zu erweitern. Folgendes Bild zeigt die logische Architektur der API:

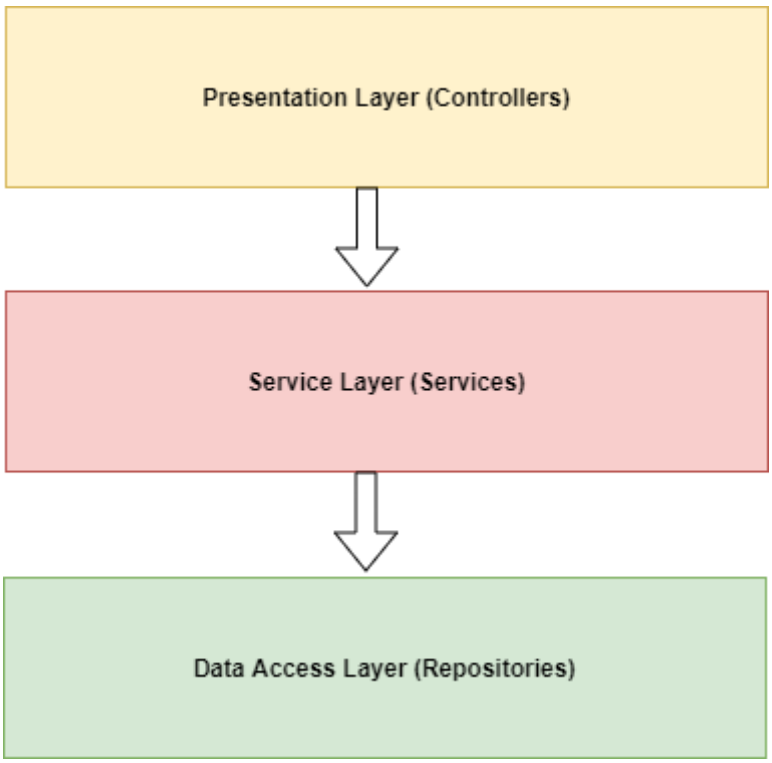


Abbildung 20: Schichtenmodell

Presentation Layer: Dies ist die Benutzeroberfläche der Anwendung, die dem Benutzer die Funktionen und Daten der Anwendung präsentiert. In der Applikation bilden die Controller-Klassen die Präsentationsschicht. Sie sind zuständig für den Empfang und der Validierung von Anfrage-Nachrichten und manipulieren das Objektmodell für die Rückgabe eines entsprechenden Data Transfer Objekts (DTO).

Service Layer: Diese Schicht enthält die Geschäftslogik, die die Kernfunktionalitäten der Anwendung steuert. Dazu gehören Entscheidungen, Berechnungen, Auswertungen und die Verarbeitung der Daten, die zwischen den anderen beiden Schichten übergeben werden.

Data Access Layer: Diese Schicht ist für die Interaktion mit Datenbanken verantwortlich, um Daten zu speichern und abzurufen. Repository-Klassen bilden in der Applikation die Datenzugriffsschicht. Die Verantwortung beschränkt sich auf CRUD-Operationen (Create, Read, Update und Delete) auf einer Datenbank.

2.2.4 Technologien

Backend-Applikation (API)

- Spring Boot für die Entwicklung der Rest API
- Spring Security für die Zugriffskontrolle mit dem OAuth2 Protokoll
- ORM mit Hibernate
- Jacoco für Code Coverage
- JUnit5 für die Umsetzung von Unit und Integrationtests
- Keycloak-Plugin zur Verbindung mit dem Authentifzierungs-Service

Authentifizierung-Service

- Keycloak wird verwendet um das Open ID Connect (OIDC) Protokoll für die Anmeldung zu verwenden und für die Erstellung eines OAuth2 konformen JWT Token.

Datenbank

- PostgreSQL & H2: Mit entsprechend eingesetzten Datenbank Drivers

API Gateway

- Tyk.io

Begründung

Einige der verwendeten Technologien wurden vom Praxispartner vorgeben. Die Verwendung von Spring Boot und der Keycloak Software sowie der Einsatz des API-Gateways Tyk wurde durch den Praxispartner bestimmt. Für die restlichen Technologien wird die Wahl mit der positiven Erfahrung aus vergangen Projekten begründet.

3 Resultat

Zielerreichung

Alle definierten Ziele konnten vollständig umgesetzt werden. Die nachfolgenden Kapitel beschreiben die implementierten Ergebnisse der einzelnen Bestandteile des entwickelten Systems.

Entwicklung der Beispielapplikation

Testing der Beispielapplikation

3.1 Entwicklung der Beispielapplikation

Ergebnis

Die Spring Boot Applikation wurde nach den funktionalen Anforderungen implementiert (vgl. Kapitel 2.1). Alle definierten Use-Cases konnten erfolgreich und vollumfänglich umgesetzt werden.

Dokumentation

Die REST-API wurde mit OpenAPI und Spring Rest Docs dokumentiert. Die Dokumentation mit Swagger wird automatisch mithilfe des OpenAPI Plugins aus dem Code generiert.

Die aus dem Code generierte Swaager Dokumentation ist unter der nachfolgenden URL einsehbar:{url}/swagger-ui/index.html?configUrl=/api-docs/swagger-config#/

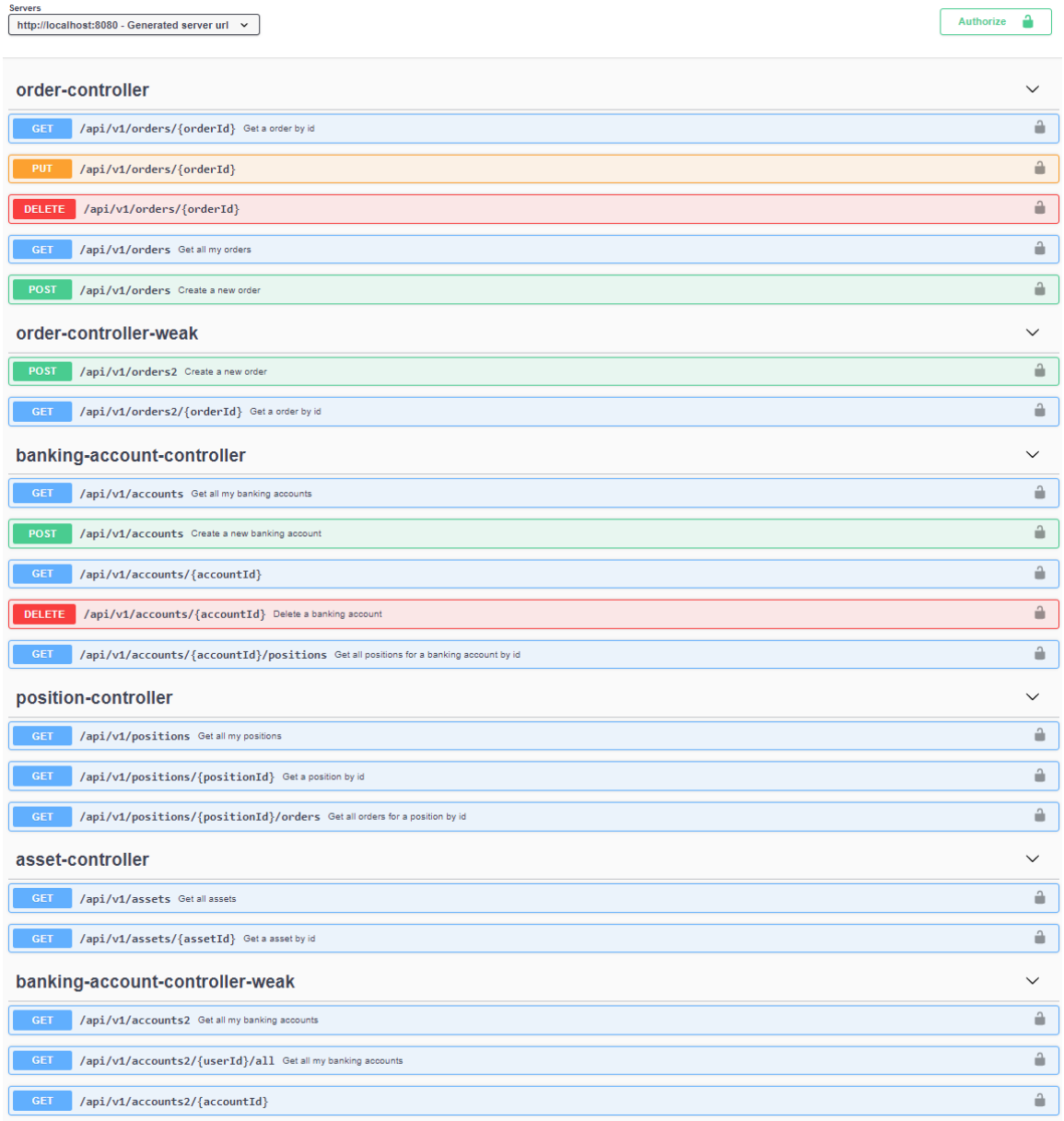


Abbildung 21: Swagger Dokumentation

Spring REST Docs ist ein Framework, das von der Spring-Community entwickelt wurde, um eine genaue Dokumentation für RESTful-APIs zu erstellen. Es verfolgt im Gegensatz zu OpenAPI einem testgetriebenen Ansatz, wobei die Dokumentation aus den geschriebenen Unittests mit Spring-MVC-Tests erstellt wird. Es kombiniert handgeschriebene Dokumentation, die mit einem Asciidoctor geschrieben wurde, mit automatisch generierten Snippets, die mit Spring MVC Test erstellt wurden. Mit RestDocs ist man weniger eingeschränkt in der Dokumentation als bei Tools wie Swagger, da man die Dokumentation manuell erweitern und so besser strukturieren kann. (vgl. [46]). Die RestDocs Dokumentation wird immer automatisch neu erstellt wenn Gradle die Unit- und Integrationtests ausführt. Dabei wird ein HTML File generiert, welches entsprechend verteilt oder auf eine Webserver hochgeladen werden kann.

In der nachfolgenden Abbildung ist ersichtlich wie ein Snippet aus einem Integrationtest gebildet wird:

```
@WithMockKeycloakAuth(authorities = {"SCOPE_account:read"},
        id = @IdTokenClaims(sub = "698eacce-f0a0-400b-8704-6d44c334b767"))
@Test
public void whenGetSingleAccount_thenSuccessful() throws Exception{
    UUID uuid = repository.findAll().get(0).getId();
    this.mockMvc.perform(get( urlTemplate: "/api/v1/accounts/{id}",uuid))
            .andDo(print())
            .andExpect(status().isOk())
            .andDo(document( identifier: "integrationTests/account/getSingle", preprocessRequest(prettyPrint())
                    pathParameters(parameterWithName("id").description("id of account to be searched"))));
```

Abbildung 22: Integrationtest mit RestDocs

Die markierte Codezeile sorgt dafür das bei der Durchführung des Tests der jeweilige Aufruf mit seiner Anfrage und der zugehörigen Antwort dokumentiert. Wenn die Anfrage Parameter entgegennimmt, können diese zusätzlich beschrieben werden.

Die generierten Snippets wurden dann in einem manuell erstellten Asciidoc zusammengefügt und werden mittels Befehl in ein ansehnliches HTML Dokument gewandelt.

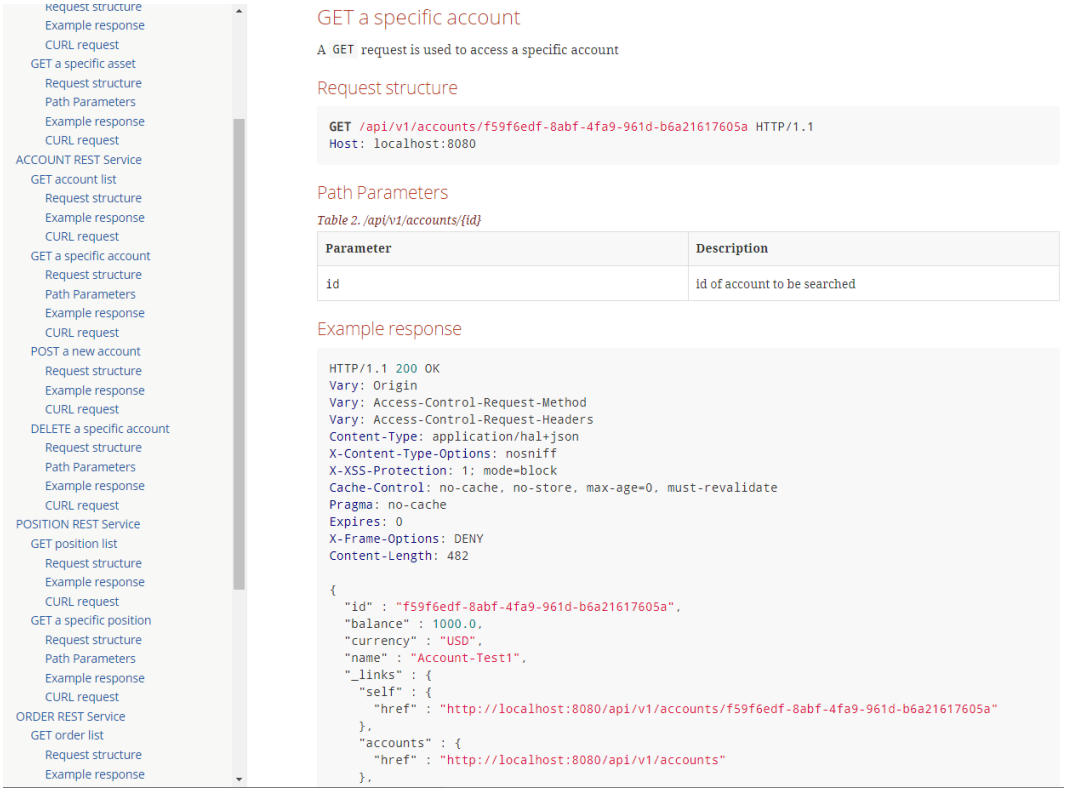


Abbildung 23: Dokumentation eines Integration Tests

Hypertext Application Language (HAL)

Hypermedia ist ein wichtiger Aspekt von REST. Er ermöglicht es, Dienste zu bauen, die Client und Server weitgehend entkoppeln und sich unabhängig voneinander weiterentwickeln lassen. Die für REST-Ressourcen zurückgegebenen Antworten (Schemas) enthalten nicht nur Daten, sondern auch Links zu verwandten Ressourcen.

Die API verwendet dafür den Media Type application/hal+json. Hypertext Application Language (HAL) ist ein Internet-Draft von der IETF für die Definition von Hypermedia.

Die Antworten der API werden im HAL ist so strukturiert, dass Elemente auf der Basis von zwei Konzepten dargestellt werden: Ressourcen und Links. Ressourcen bestehen aus Links, eingebetteten Ressourcen und ihren eigenen Attributen. Links haben einen

Ziel-URI sowie den Namen des Links. In der nachfolgenden Abbildung wird OrderResponse Nachricht der API angezeigt:

Example response

```
HTTP/1.1 200 OK
Content-Type: application/hal+json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Length: 965

{
  "status" : "PLACED",
  "id" : "e066e29f-c2dd-4a1a-880c-f3f99a2f6fc4",
  "type" : "BUY",
  "quantity" : 100,
  "price" : 120.23,
  "asset" : {
    "id" : 1,
    "symbol" : "AMZN",
    "currency" : "USD"
  },
  "positionId" : "ed0d9a8c-d50a-4ed1-b061-e09351e046c1",
  "accountId" : "e8650c75-508c-4ebe-99b7-cf1ca4d3c742",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/api/v1/orders/e066e29f-c2dd-4a1a-880c-f3f99a2f6fc4"
    },
    "orders" : {
      "href" : "http://localhost:8080/api/v1/orders"
    },
    "position" : {
      "href" : "http://localhost:8080/api/v1/positions/ed0d9a8c-d50a-4ed1-b061-e09351e046c1"
    },
    "positions" : {
      "href" : "http://localhost:8080/api/v1/positions"
    },
    "account" : {
      "href" : "http://localhost:8080/api/v1/accounts/e8650c75-508c-4ebe-99b7-cf1ca4d3c742"
    },
    "accounts" : {
      "href" : "http://localhost:8080/api/v1/accounts"
    }
  }
}
```

Attribute

Links

Abbildung 24: OrderResponse mit HAL

Die OrderResponse enthält Attribute und Links. Eine eingebettete Ressource ist nicht vorhanden. Die Links enthalten URI-Links, welcher der Benutzer bei Bedarf verwenden kann, um ausführlichere Informationen zu einer zugehörigen Ressource zu erhalten.

3.1.1 Datenbank

Ergebnis

Als Datenbanksystem wird PostgreSQL genutzt. Während der Entwicklung und für die Integrationstests wurde eine H2 In-Memory Datenbank verwendet. Die Erstellung des Datenbankschemas erfolgt bei jedem Start der Applikation von neuem. Die Datenbank wird beim Starten der Applikation mittels einer Konfigurationsdatei mit Beispieldaten befüllt. Die Objekterelation wurde mit dem ORM-Mapper Hibernate umgesetzt.

3.1.2 Verwendung von UUID

Resultat

Ein Universally Unique Identifier (kurz UUID) repräsentiert eine eindeutige Zeichenfolge. Eine UUID besteht aus hexadezimalen Ziffern (je 4 Zeichen) und 4 "-" Symbolen, so dass ihre Länge 36 Zeichen entspricht. Die Standarddarstellung der UUID verwendet die hexadezimale Schreibweise:

```
"id": "97f20074-ec67-455b-b164-f6f9acf2dd39",
```

Abbildung 25: Beispiel UUID

Die API verwendet für alle Objekte eines Benutzers, eine solche UUID. Die UUID dient als eindeutige Identifikation eines Objektes in der Datenbank. Der Einsatz von UUIDs wird von OWASP als Massnahme für die häufigste Schwachstelle (Broken Object Level Authorization) empfohlen. Durch UUIDs ist es für einen Angreifer nicht möglich die ID eines fremden Objektes zu erraten. Die Ausnutzung der Schwachstelle wird dadurch für einen Angreifer stark erschwert.

```
GET /api/v1/accounts/f59f6edf-8abf-4fa9-961d-b6a21617605a
```

Abbildung 26: Request mit UUID

Intern wird eine UUID Version 4 (RFC 4122) verwendet, was bedeutet das zufällige Zahlen für die Erstellung der UUID verwendet werden. Die Erstellung der UUID erfolgt mit Hibernate und gilt als Secure Random, da sie einen nicht vorhersagbaren Wert als Seed für die Generierung von Zufallszahlen verwendet und so die Wahrscheinlichkeit von Kollisionen verringert. Das nachfolgende Bild zeigt wie mit Hibernate automatisch eine UUID bei der Erstellung eines neuen Banking Accounts generiert wird:

```
public class BankingAccount {
    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(
            name = "UUID",
            strategy = "org.hibernate.id.UUIDGenerator"
    )
    @Column(name = "account_id", updatable = false)
    private UUID id;
```

Abbildung 27: Hibernate UUID Erstellung

3.1.3 Validierung von Parametern

Ergebnis

Bei der Entwicklung der API wurde für jede Anfrage und jede Response Nachricht ein eigenes Data Transfer Object (DTO) erstellt. Für Objekte, welche im Body einer http Anfrage an die API gesendet werden, wurden exemplarische Validierungen eingebaut. Die Validierung wurde mit der Standard-Validierungsspezifikation “Bean Validation“ (vgl. [47]) umgesetzt, diese ermöglicht es Objekte auf einfache Weise zu validieren, indem für die Einschränkungen in Form von Annotations deklariert werden. Die Validierung selbst wird durch den zuständigen Controller durchgeführt. Nachfolgendes Bild zeigt exemplarisch das Anfrage- und Antwort-Objekt eines Banking Accounts an.

```java
public class BankingAccountRequest {
    @JsonProperty("balance")
    @NotNull
    @Min(0)
    @Max(100000)
    private Double balance;

    @NotBlank
    @Size(min= 3,
            max=3, message = "invalid length for c
    private String currency;

    @NotBlank
    @JsonProperty("name")
    private String name;

    @JsonProperty
    @Size(min=36,max=36, message = "invalid format
    private String userId;
```

Abbildung 28: BankingAccountRequest

```java
public class BankingAccountResponse {
    @JsonProperty("id")
    private UUID id;

    @JsonProperty("balance")
    private Double balance;
    @JsonProperty("currency")
    private String currency;
    @JsonProperty("name")
    private String name;
```

Abbildung 29: BankingAccountResponse

Bei der Verwendung von DTOs sind immer Mapper nötig, um das Data Transfer Objekt intern zu der Objekt-Entity zu wandeln. Wenn diese Mapper falsch konfiguriert sind, besteht die Gefahr einer Mass Assigment Schwachstelle. Für das Mapping wurde die Bibliothek von MapStruct [48] verwendet.

.

3.1.4 Einbau von Schwachstellen

Ergebnis

Für die API wurden für zwei Controller implementiert, welche bewusst Schwachstellen der OWASP Top 10 Liste enthalten. Die nachfolgende Liste zeigt, welche Schwachstellen implementiert worden sind:

Banking Account Controller Weak

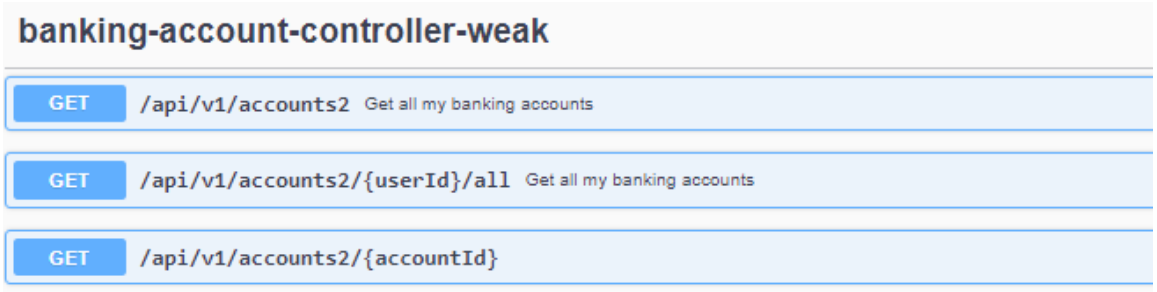


Abbildung 30: Banking Account Controller Weak

Methode	URI	Schwachstelle
GET	{url}/api/v1/account2/	SQL Injection (Request Parameter)
GET	{url}/api/v1/account2/{userId}/all	SQL Injection (URL Path)
GET	{url}/api/v1/account2/{accountId}	Broken Object Level Authorization

Order Controller Weak

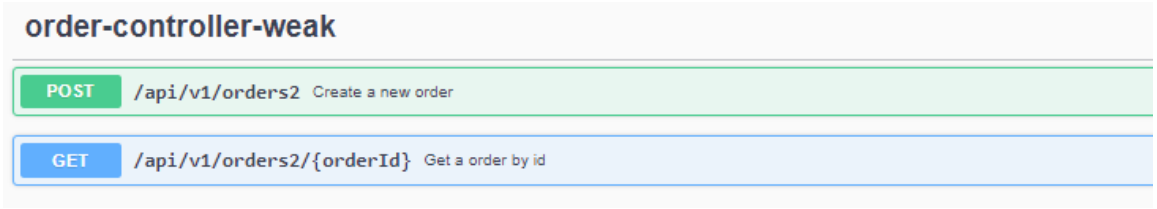


Abbildung 31: Order Controller Weak

Methode	URI	Schwachstelle
POST	{url}/api/v1/orders2/	Mass Assigment
GET	{url}/api/v1/orders2/{orderId}	Excessive Data Exposure

Falsche Konfiguration der CSP

Die CSP wurde bewusst falsch konfiguriert, um eine weitere Schwachstelle in die API einzubauen. Die Swagger-Dokumentation verwendet Javascript um das User-Interface anzuzeigen. Wenn die Content-Security-Policy (CSP) auf „script-src: self“ gestellt wird

Ergebnis

kann die Dokumentation nicht mehr geladen werden. Dies wird in der nachfolgenden Abbildung gezeigt.

```
http.headers().xssProtection().and().contentSecurityPolicy( policyDirectives: "script-src 'self'");
```

Abbildung 32: CSP mit script-src: self

```
Refused to execute inline script      :8080/swagger-ui/ind…swagger-config#/:38
because it violates the following Content Security Policy directive: "script-
src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-
XxTFtQeaV2+Qbm2I5xr2m6z7I3WRujCeoPc6KgnzUrA='), or a nonce ('nonce-...') is
required to enable inline execution.
```

Abbildung 33: Swagger Fehlermeldung

Ein Entwickler, der diesen Fehler erhält und sich nicht mit der Konfiguration einer CSP auskennt könnte auf die Idee kommen „unsafe-inline“ zu verwenden, um den Fehler zu beheben. Das Verwenden von „unsafe-inline“ in einer CSP wird als grosses Sicherheitsrisiko angesehen (vgl. [49]). Korrekterweise müsste die CSP den Hash oder die Nonce des Javascripts verwenden. Um jedoch zu überprüfen, ob die falsche Konfiguration durch eine statische Code Analyse oder anderen Tools entdeckt wird, wurde „unsafe-inline“ verwendet.

```
http.headers().xssProtection().and().contentSecurityPolicy( policyDirectives: "script-src 'self' 'unsafe-inline'");
```

Abbildung 34: CSP mit unsafe-line

Mit den Schwachstellen soll überprüft werden ob, diese durch manuelle oder automatische Tests erkannt werden können.

3.1.5 Authentifizierung

Resultat

Die Authentifizierung wurde mit der Keycloak Anwendung umgesetzt und verwendet ein sogenanntes standarisiertes JSON Web Access Token, kurz JWT. Das Format eines JWT Tokens wird im RFC 7519 genau (vgl. [50]) beschrieben. Das Problem mit dem JWT ist, dass es lesbare Informationen wie z.B. die Benutzer-ID beinhaltet. Jeder kann ein JWT-Token entschlüsseln und die darin enthalten Informationen auslesen. Die nachfolgende Abbildung zeigt wie sich ein solches Token auf jwt.io entschlüsseln lässt:

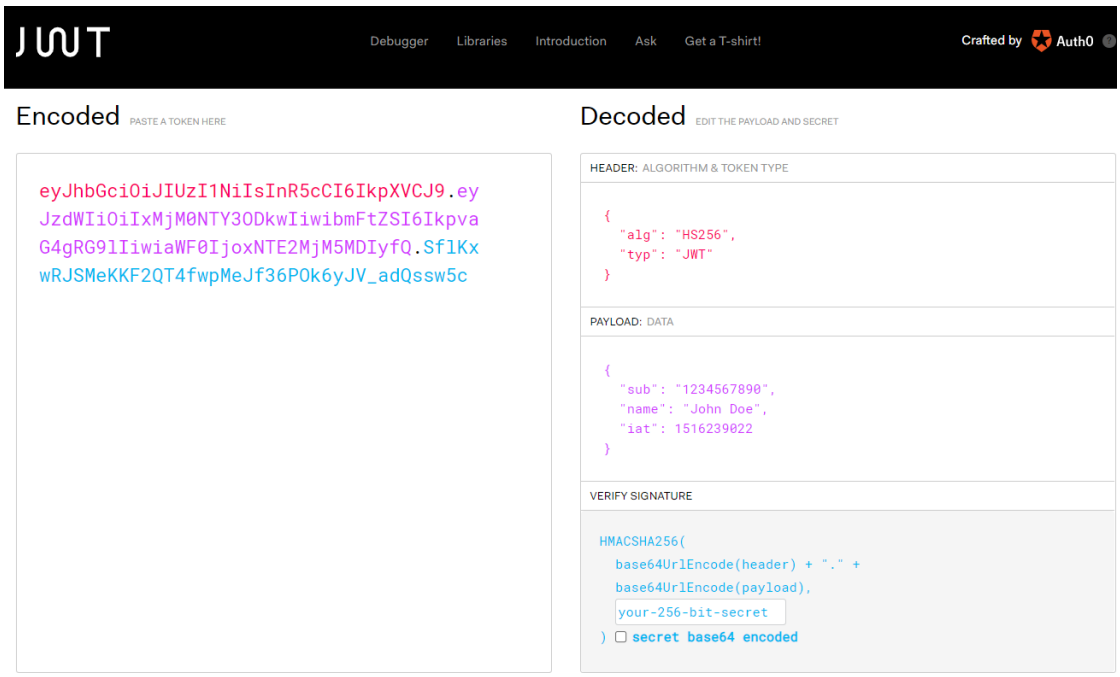


Abbildung 35: Token entschlüsselt auf jwt.io

Gestohlene Tokens können so von jedem ausgelesen werden und von einem Angreifer ausgenutzt werden. Ein JWT wird immer in drei Bereiche unterteilt. Der Header besteht in der Regel aus zwei Teilen: dem Typ des Tokens, also JWT, und dem verwendeten Signieralgorithmus, z. B. HMAC SHA256 oder RSA. In der Vergangenheit wurden Angriffe durchgeführt, welche beispielweise erfolgreich den Signieralgorithmus auf „none“ gestellt haben, um so dass Token fälschen zu können. Der zweite Teil des Tokens sind die Payload-Daten, die die Claims enthält. Claims sind Aussagen über eine Entität (typischerweise der Benutzer) und zusätzliche Daten (vgl. [51]). Wie in der obigen Abbildung ersichtlich können auch personenbezogene Daten, wie der Name des Benutzers enthalten sein.

Resultat

Abbildung 36: JWT-Token vom Keycloak Server

Die obige Antwort enthält ein JWT Access Token, welches vom eingesetzten Keycloak Server ausgegeben wurde. Neben dem Access Token wurde auch der Refresh Token mitgeschickt. Für das Access Token wurden diverse Berechtigungen erteilt, welche im Scope Attribut in der obigen Antwort ersichtlich sind.

Um zu verhindern, dass diese Informationen durch einen Angreifer gestohlenen werden können, wurde der Split-Token Ansatz (vgl. [52]) mit einem JavaScript entwickelt und angewendet. Dieser Ansatz schlägt vor, nur die Signatur des JWT-Zugriffstokens auf der Client-Seite zu verwenden und den Header und die Claims des JWTs auf der Server-Seite zu speichern. Anstelle eines lesbaren JWTs erhält der Client die Signatur in Form eines sogenannten Opaque Access Tokens. Ein Opaque Token hat im Gegensatz zu

einem JWT ein Format, welches nur dem Ersteller bekannt ist und somit nicht für jedermann lesbar ist.
Wenn ein Client sich beim Gateway Endpoint /auth/login anmeldet leitet dieser die Anmeldedaten des Benutzers an den Keycloak Server. Bei korrekten Anmeldedaten erhält der API-Gateway ein JWT Access Token für den Zugriff auf die entsprechende API.
Er speichert das JWT Token in seiner Datenbank und erstellt einen zugehörigen Key aus der Signatur des Tokens. Der API-Gateway liefert diesen Key in Form eines Opaque-Tokens an den Client aus.

Abbildung 37: Anfrage an Gateway für Opaque Token

Die Antwort des Gateways anstelle des kompletten JWT Access Tokens nur noch die Signatur des Tokens. Der Refresh Token wurde entfernt da dieser nicht verwendet wird. Die Berechtigungen (Scopes) sind bereits im kompletten JWT-Token bereits enthalten und müssen deshalb auch nicht an den Client gesendet werden.

Der nachfolgende Codeausschnitt zeigt auf, wie dies mit Hilfe eines JavaScripts umgesetzt worden ist:

```
// Parse the response
var bodyObj = JSON.parse(responseObj.Body);
var accessTokenComplete = bodyObj.access_token;
var signature = accessTokenComplete.split(".")[2];

createOpaqueTokenInTyk(accessTokenComplete)

//create response to client
bodyObj.access_token = signature;
delete bodyObj.refresh_expires_in;
delete bodyObj.refresh_token;
delete bodyObj.scope
```

Abbildung 38: Konfiguration der Antwort

Die Funktion createOpaqueTokenInTyk erstellt einen Opaque Token (Key) aus der Signatur des Access Tokens, mit einer Referenz auf das komplette Access Token. Die nachfolgende Abbildung zeigt einen solchen Opaque Token in Tyk:

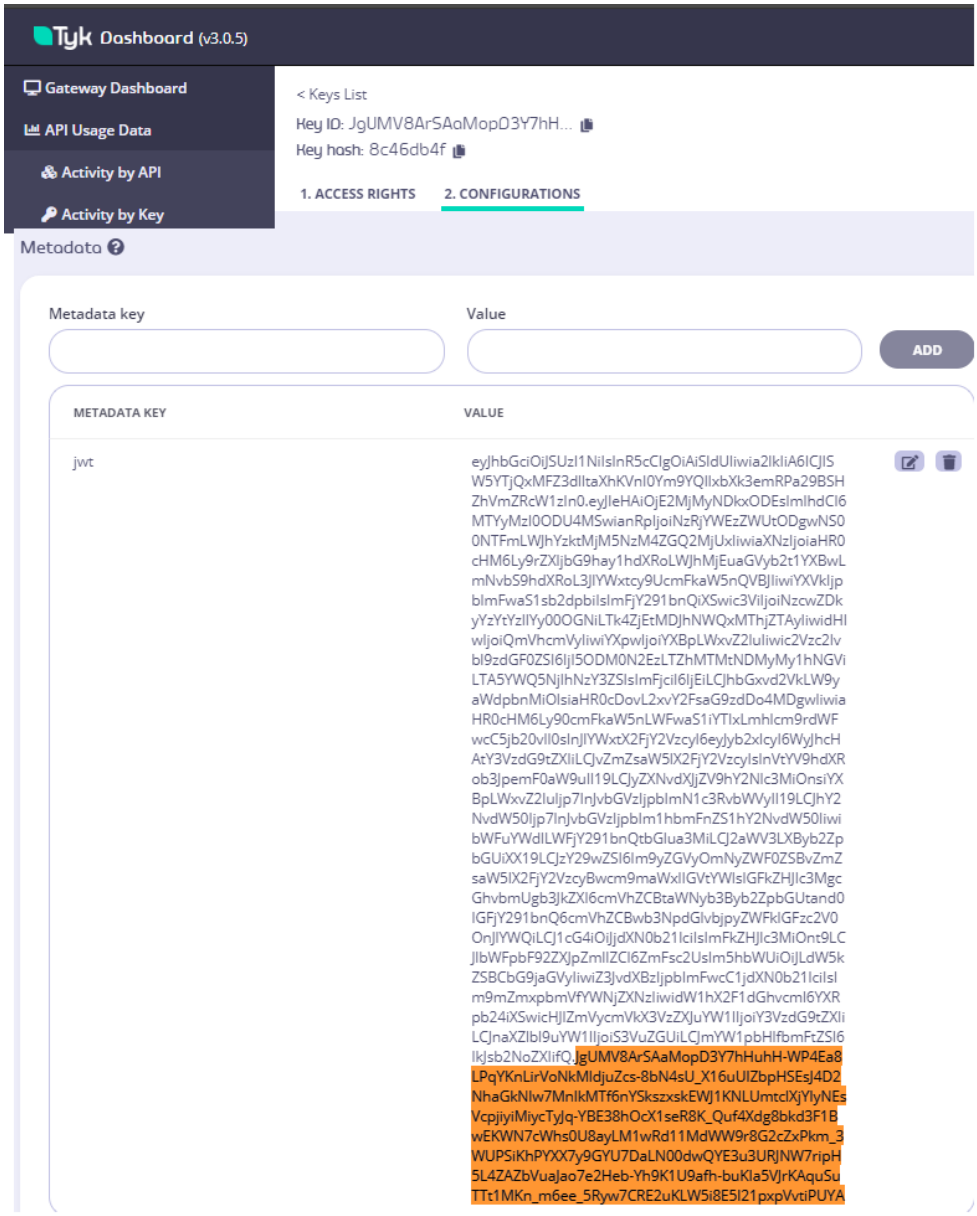


Abbildung 39: Opaque Token Tyk

Die Key-ID besteht aus der Signatur des JWTs und ist in der Abbildung markiert. In den Metadaten des Keys wird das komplette JWT gespeichert. Wenn der Client eine Anfrage sendet, nimmt der API-Gateway den vom Client gesendeten Opaque-Token (Signaturteil) und sucht ihn seinem Speicher nach dem zugehörigen JWT Token. Er erweitert die Anfrage dann mit dem zugehörigen JWT, bevor er sie an den jeweiligen API-Dienst weiterleitet. Auf diese Weise erhält die API ein vollständiges JWT, das je nach Bedarf verwendet werden kann.

Abbildung 40: Client-Anfrage an Gateway mit Opaque Token

Die obige Abbildung zeigt die Anfrage mit dem Opaque Token an den API-Gateway mit Das Token ist viel kürzer geworden, da es nur die Signatur enthält. In der nach

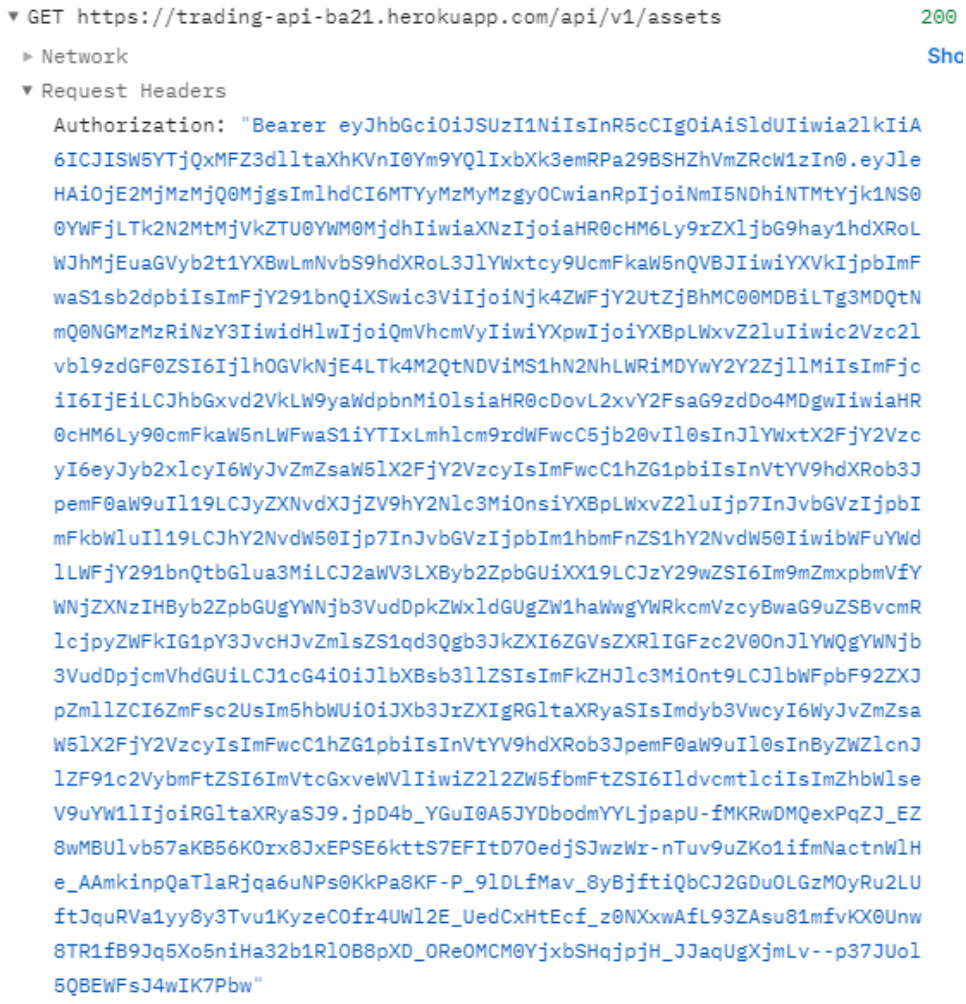


Abbildung 41: API-Gateway leitet Anfrage mit JWT an API weiter

Dieser Ansatz bietet sowohl für den Client als auch für das Backend Vorteile. Wir erhalten die Flexibilität von JWTs durch die Möglichkeit, Benutzerinformationen in Claims zu speichern und haben gleichzeitig die Sicherheit eines Opaque-Tokens, da nicht das gesamte JWT Token an den Client gesendet wird.

3.1.6 Autorisierung

Resultat

Die Autorisierung wurde mit Spring Security und OAuth2 erfolgreich umgesetzt. Der Ressourcenserver ist ein OAuth 2.0-Begriff und stellt in unserem System die Spring-Boot-Applikation dar. Der Ressourcenserver bearbeitet authentifizierte Anfragen, nachdem die Anwendung ein Zugriffstoken erhalten hat. Der Ressourcenserver erhält Anfragen von Anwendungen mit einem HTTP-Autorisierungs-Header, der ein Access Token enthält. Er überprüft das Zugriffstoken, um festzustellen, ob eine Anfrage verarbeitet werden darf.

Validierung des Access Tokens

Die Validierung des Access Tokens wurde so umgesetzt, wie es von OAuth2 vorgeschlagen wird (vgl. [53]) Ein Zugriffstoken ist für eine API bestimmt und sollte demzufolge nur von der API validiert werden, für die es bestimmt ist. Für die Validierung des Access Tokens wurden die nachfolgenden Prüfungen umgesetzt:

1. **Standard JWT Validierung:** Da das Access Token ein JWT Token ist wurden überprüft, ob der korrekte Algorithmus für die Signierung verwendet wird. Für die Überprüfung der Signatur verwendet die Beispielapplikation den RSA Public Key des Keycloak-Servers. Dieser wird über eine eindeutige URI ausgelesen und verifiziert die Signatur des Tokens. So kann sichergestellt werden, dass das Token nicht durch jemanden verändert wurde. Der Ressource-Server überprüft auch die Gültigkeitsdauer (Expiration) des Tokens und den Ersteller (Issuer) des Tokens.

2. **Überprüfen der Token Audience:** Die Beispielapplikation überprüft, ob das JWT-Token den korrekten Client-Namen (Ressource) als Audience enthält. Die Beispiel-API akzeptiert nur Tokens mit der Audience „api-login“. Für andere Clients wird der Zugriff abgelehnt.

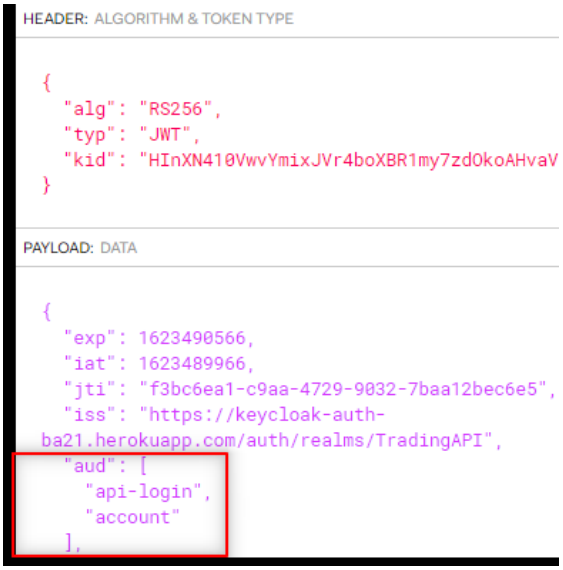


Abbildung 42: JWT mit Audience api-login.

3. **Überprüfen von Berechtigungen (Scopes):** Für die Steuerung der Zugriffskontrolle wurde OAuth2 Scope Mechanismus verwendet (vgl. [54]). Der nachfolgende Ausschnitt aus dem Keycloak Server zeigt auf, welche Scopes bei dieser Arbeit verwendet wurden:

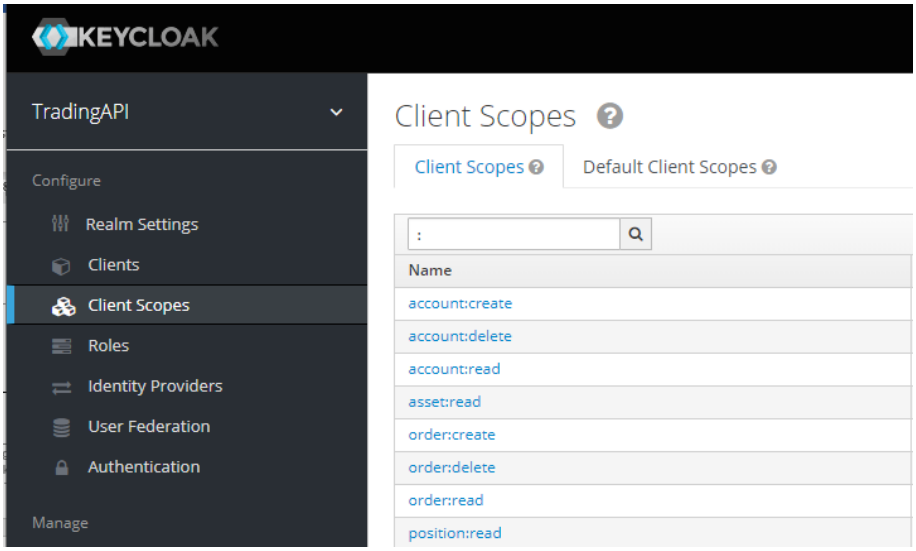


Abbildung 43: Eingesetzte Scopes

Für die Beispielapplikation wurden die Rollen Administrator und Customer definiert. Die Scopes wurden wie folgt den jeweiligen Rollen zugeordnet:

Rolle Administrator	Rolle Customer
account:create, account:delete, asset:read, order:delete	account:read, asset:read, order:create, order :read, position:read

Die Zugriffsrechte wurden nach Create, Read und Delete aufgeteilt und pro Ressource aufgeteilt. Der Client wird bei der Authentifizierung gezwungen die entsprechend benötigten Scopes anzugeben. Wenn er keine Berechtigung für den Scope besitzt, erhält er diesen nicht. In der Spring Boot Applikation wurde mit Spring Security die nachfolgende Konfiguration aufgesetzt:

```
http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api-docs").permitAll()
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/v3/api-docs/**").permitAll()
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/swagger-ui/**").permitAll()
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/swagger-ui.html").permitAll()
        .antMatchers(HttpMethod.GET, ...antPatterns: "/api/v1/assets").hasAuthority("SCOPE_asset:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/assets/{id}").hasAuthority("SCOPE_asset:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/accounts").hasAuthority("SCOPE_account:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/accounts/{id}").hasAuthority("SCOPE_account:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/accounts2/{id}").hasAuthority("SCOPE_account:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/accounts2").hasAuthority("SCOPE_account:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/accounts2/{id}/all").hasAuthority("SCOPE_account:read")
        .antMatchers(HttpMethod.POST, ...antPatterns: "/api/v1/accounts").hasAuthority("SCOPE_account:create")
        .mvcMatchers(HttpMethod.DELETE, ...mvcPatterns: "/api/v1/accounts/{id}").hasAuthority("SCOPE_account:delete")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/accounts/{id}/positions").hasAuthority("SCOPE_position:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/positions").hasAuthority("SCOPE_position:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/positions/*").hasAuthority("SCOPE_position:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/positions/{id}/orders").hasAuthority("SCOPE_order:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/orders").hasAuthority("SCOPE_order:read")
        .mvcMatchers(HttpMethod.GET, ...mvcPatterns: "/api/v1/orders/{id}").hasAuthority("SCOPE_order:read")
        .antMatchers(HttpMethod.POST, ...antPatterns: "/api/v1/orders").hasAuthority("SCOPE_order:create")
        .antMatchers(HttpMethod.POST, ...antPatterns: "/api/v1/orders2").hasAuthority("SCOPE_order:create")
        .mvcMatchers(HttpMethod.DELETE, ...mvcPatterns: "/api/v1/orders/{id}").hasAuthority("SCOPE_order:delete")
        .mvcMatchers(HttpMethod.PUT, ...mvcPatterns: "/api/v1/orders/{id}").hasAuthority("SCOPE_order:delete")
        .anyRequest() ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
        .denyAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
        .and() HttpSecurity
        .oauth2ResourceServer() OAuth2ResourceServerConfigurer<HttpSecurity>
        .jwt();
```

Abbildung 44: Security Konfiguration Spring Boot

Mit dem Befehl. denyAll() werden alle Zugriffe auf die API automatisch abgelehnt. Dies wird auch von OWASP empfohlen, um die API-Schwachstelle Nr 5. Broken Function Level Authorization abzumindern. Für alle API-Endpoints muss der Entwickler so bewusst, die Zugriffskontrolle einrichten, da ansonsten der Zugriff abgelehnt wird.

Broken Object Level Authorization

Ein Kunde darf nicht auf Applikationsdaten eines anderen Kunden zugreifen. Da die Beispielapplikation selbst keine User-Informationen speichert wurden die Informationen aus dem JWT Access Token verwendet, um zu überprüfen, ob ein Benutzer mit korrekten Berechtigungen (Scope) auch der Besitzer der jeweiligen Ressource ist.

Die nachfolgende Abbildung zeigt den Endpoint /accounts/{id} des Banking Account Controllers. Der Endpoint ist nur für Benutzer mit dem Scope „account:read“ zugänglich.

```
@GetMapping(path = "{accountId}")
public EntityModel<BankingAccountResponse> getSingleAccount(@PathVariable("accountId") UUID accountId) {
    BankingAccountResponse account = mapstructMapper.bankingAccountToBankingAccountResponse(
            bankingAccountService.getAccount(getTokenSubject(),accountId));

    return assembler.toModel(account);
}
private String getTokenSubject() { return getKeycloakSecurityContext().getToken().getSubject(); }
```

Abbildung 45: Endpoint /accounts/{id}

Ein Banking Account besitzt immer eine User-ID, welche bei der Abfrage eines spezifischen Accounts nebst der Account-ID immer benötigt wird. Die User-ID eines Benutzers befindet sich im Subject des JWT-Tokens.

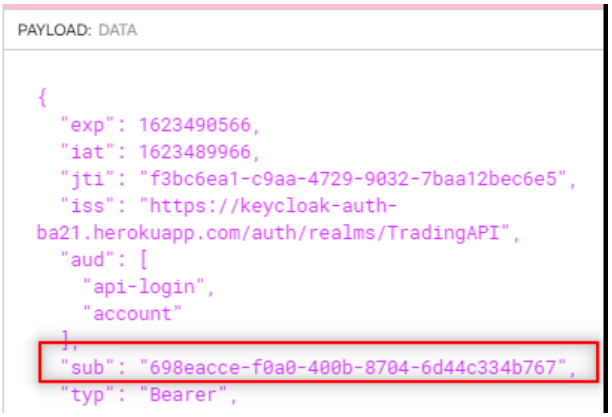


Abbildung 46: Subject eines JWT-Tokens

Wenn ein Benutzer eine Anfrage für eine Account-ID stellt, welche nicht seinem User zugeordnet ist, antwortet die API mit dem Status Code 404 (Not Found).

3.2 Testing der Beispielapplikation

Inhalt

Das Kapitel befasst sich mit dem Testing der Beispielapplikation und es werden die nachfolgenden Test-Arten beschrieben:

- Automatisierte Unit- und Integration-Tests
- End-to-End Security Tests
- Statische Code Analyse
- Dynamische Sicherheitstests

3.2.1 Automatisierte Unit- und Integration-Tests

Ergebnis

Die automatisierten Unit- und Integration-Tests wurden mit JUnit 5 (vgl. [33]) geschrieben. Die benötigten Mocks wurden mit Spring MVC umgesetzt. Die Test Coverage wird mit Jacoco (vgl. [34]) ausgewertet. Die Tests werden automatisch bei jedem Commit durch die CI/CD Pipeline ausgeführt. Der Build-Prozess der Spring Boot Applikation wird unterbrochen, wenn ein Test fehlschlägt.

Testabdeckung

Im CI/CD Prozess wird eine Testabdeckung in Prozent mittels der Java Code Coverage Libary (Jacoco) [55] ermittelt und angezeigt. So ist die Testabdeckung nach jedem Gitlab Commit ersichtlich.

```
----- Code Coverage ----------
 - instruction:  72.83%
 - branch     :  26.34%
 - line       :  74.42%
 - complexity :  58.37%
 - method     :  72.49%
 - class      :  88.89%
------------------------------
```

Abbildung 47: Jacoco Test Coverage Gitlab CI/CD

Zusätzlich wird der ausführliche HTML Report von Jacoco als Artefakt zum Download in der CI/CD Pipeline bereitgestellt.

SecureTradingAPI

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed
com.lgt.dapi.api.SecureTradingAPI.domain.entities		36%		2%	79	130	53	142	42
com.lgt.dapi.api.SecureTradingAPI.controller.mapper		59%		43%	41	65	90	209	7
com.lgt.dapi.api.SecureTradingAPI.controller		70%		43%	15	46	49	170	8
com.lgt.dapi.api.SecureTradingAPI.service		62%		36%	26	49	31	100	16
com.lgt.dapi.api.SecureTradingAPI.controller.dtos		87%		n/a	12	104	14	126	12
com.lgt.dapi.api.SecureTradingAPI.service.exception		64%		n/a	8	24	12	32	8
com.lgt.dapi.api.SecureTradingAPI.config		99%		n/a	2	14	2	183	2
com.lgt.dapi.api.SecureTradingAPI		37%		n/a	1	2	2	3	1
com.lgt.dapi.api.SecureTradingAPI.controller.assembler		100%		n/a	0	8	0	24	0
Total	1'399 of 5'149	72%	137 of 186	26%	184	442	253	989	96

Abbildung 48: Jacoco HTML Report

Das festgelegte Ziel von mindestens 60% Coverage konnte mit 72% Coverage übertroffen werden.

Security Testing

Die Möglichkeit, Integrationstests auszuführen, ohne eine eigene Integrationsumgebung zu benötigen, ist ein wertvolle Eigenschaft für jede Applikation. Die Verwendung von Spring Security macht es einfach, geschützte Ressourcen zu testen. Durch die Annotation @WithMockKeycloakAuth kann ein Access Token mit den nötigen Informationen für einen Test erstellt werden. Dadurch wird die mit den Integration Tests die korrekte Implementation der Zugriffskontrolle überprüft. Der nachfolgende Test überprüft die korrekte Funktionalität, wenn ein autorisierter Benutzer mit der korrekten Berechtigung auf den End Point /api/v1/orders zugreift:

```
@Test
@WithMockKeycloakAuth(authorities = {"SCOPE_order:read"},
        id = @IdTokenClaims(sub = "698eacce-f0a0-400b-8704-6d44c334b767"))
public void whenGetAllOrders_thenSuccessfulCustomer() throws Exception {
    Order myOrder = repository.findAllOrdersByUser( userId: "698eacce-f0a0-400b-8704-6d44c334b767").get().get(0);
    Order orderOtherUser = repository.findAllOrdersByUser( userId: "770d92c6-c9ec-48cb-98f1-02a5d118ce02").get().get(0);
    ResultActions result = this.mockMvc.perform(get( urlTemplate: "/api/v1/orders/"))
            .andDo(print())
            .andExpect(status().isOk())
            .andDo(document( identifier: "integrationTests/order/getAllSuccessCustomer",
                    preprocessResponse(prettyPrint())));
    assertTrue(result.andReturn().getResponse().getContentAsString().contains(myOrder.getId().toString()));
    assertFalse(result.andReturn().getResponse().getContentAsString().contains(orderOtherUser.getId().toString()));
    assertTrue(result.andReturn().getResponse().getContentType().contentEquals( cs: "application/hal+json"));
```

Abbildung 49: Test /api/v1/orders

Der nachfolgende Tests zeigt wie die Schwachstelle Broken Object Level Authorisation getestet wurde:

```
@Test
@WithMockKeycloakAuth(authorities = {"SCOPE_order:read"},
        id = @IdTokenClaims(sub = "770d92c6-c9ec-48cb-98f1-02a5d118ce02"))
public void whenGetSingleOrder_BrokenObjectLevelAuthorization() throws Exception{
    UUID uuid = repository.findAll().get(0).getId();
    ResultActions result = this.mockMvc.perform(get( urlTemplate: "/api/v1/orders/{id
            .andDo(print())
            .andExpect(status().isNotFound())
            .andDo(document( identifier: "integrationTests/order/getSingleOrderNotFoun
                    pathParameters(parameterWithName("id").description("id of ord
    assertEquals(OrderNotFoundException.class, result.andReturn().getResolvedExce
    assertTrue(result.andReturn().getResolvedException().getMessage()
            .contains("Could not find Order with id: " + uuid));
```

Abbildung: Integration Test Broken Object Level Authorization

Mit dem Mock wird ein Token für den User mit der ID "770d92c6-c9ec-48cb-98f1-02a5d118ce02" simuliert. Im Test wird nun auf eine Order eines anderen Users zugegriffen. Die API wurde so entwickelt, dass eine OrderNotFoundException und der HTTP Status Code 404 (Not Found) zurückgegeben wird. Mit diesem Test konnte gezeigt werden das die Broken Object Level Authorisation schon beim Schreiben von Integration-Tests überprüft werden kann.

Es wurden auch Tests mit falschen Inputdaten durchgeführt. Der nachfolgende Test zeigt dies exemplarisch auf. Er prüft wie sich der End Point, verhält wenn anstelle einer

korrekten UUID ein String gesendet wird. Die API antwortet mit dem HTTP Status Code 400 (Bad Request) und gibt die nachfolgende Fehlermeldung an den Benutzer:

```
@WithMockKeycloakAuth(authorities = {"SCOPE_order:read"})
@Test
public void whenGetSingleOrder_thenBadRequest() throws Exception{
    ResultActions result = this.mockMvc.perform(get( urlTemplate: "/api/v1/orders/{id}", ...urlVariables: "1"))
            .andDo(print())
            .andExpect(status().isBadRequest())
            .andDo(document( identifier: "integrationTests/order/getSingleBadRequest", preprocessRequest(prettyPrint())
                    pathParameters(parameterWithName("id").description("id of position to be searched"))));
    assertEquals(MethodArgumentTypeMismatchException.class, result.andReturn().getResolvedException().getClass());
    assertTrue(result.andReturn().getResolvedException().getMessage()
            .contains("Invalid UUID string: 1"));
}
```

Abbildung 50: Integration Test falschem Parameter-Typ

Die obige Fehlermeldung, welche dadurch ausgelöst wird, birgt ein Sicherheitsrisiko, da sie die Eingabe des Benutzers in der Antwort reflektiert. Dies könnte zu einer erfolgreichen Reflected-XSS Angriff zur Folge haben. Der nachfolgende manuell durchgeführte Test mit Postman veranschaulicht dies:

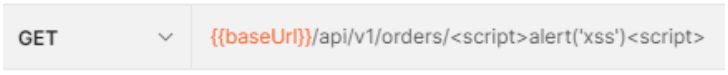


Abbildung 51: Anfrage mit Script-Tag

Abbildung 52: Antwort reflektiert die Eingabe

Die Eingabe des Benutzers wird in der Antwort 1 zu 1 reflektiert. Wenn eine Webapplikation nun die Fehlermeldung in einem HTML anzeigt, wird, ohne den Input zu validieren könnte der Angriff erfolgreich sein. Mit einer falsch konfigurierten oder nicht existenten Content Security Policy (CSP) wird das Skript dann im Kontext des Opfers ausgeführt und kann dadurch auf sensible Informationen wie z.B. das Access Token zuzugreifen. Das Access Token befindet sich bei modernen Single Page Applikationen im lokalen Speicher und kann mit einem XSS Angriff ausgelesen und im schlimmsten Fall an den Angreifer gesendet werden (vgl. [56]).

3.2.2 End-to-End Security Tests

Inhalt

Für die End-to-End Security Tests wurde das Tool Postman evaluiert. Mit Postman sollen die priorisierten Schwachstellen der Bedrohungsmodellierung getestet werden Die Tests sollen automatisch von der CI/CD Pipeline ausgeführt werden, nachdem die API bereitgestellt wurde.

Zu beachten ist, dass es sich bei den folgenden Tests um White-Box-Sicherheitstests handelt, da sie sie Vorkenntnisse bezüglich der Benutzeranmeldeinformationen und der API-Struktur erfordern (siehe Kapitel 1.7.4 API Security Testing)

Die nachfolgenden Tests wurden umgesetzt und werden in diesem Kapitel behandelt:

- Autorisierungs-Tests
- Schema Validierungs Tests
- Value Tests
- Security Header Tests
- SQL Injection Tests

Am Ende des Kapitels wird die Automatisierung der Tests beschrieben und der finale Report vorgestellt.

Postman Collection

Für die lokalen und die produktiven Tests jeweils eine Collection erstellt.

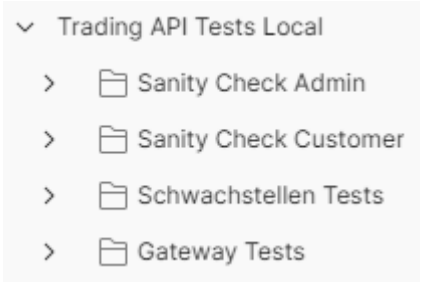


Abbildung 53: Lokal Test Collection

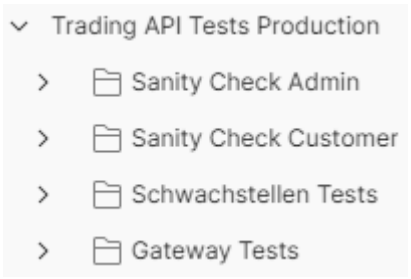


Abbildung 54: Produktive Test Collection

Die Sanity Check Collections überprüfen die korrekte Funktionsweise der API und Schwachstellen der API. Die Schwachstellen Test Collection testet die eingebauten Schwachstellen der API.Die Test in dieser Collection sollen fehlschlagen, um aufzuzeigen das diese durch Test erkannt wurden. Die Tests werden direkt auf der API ausgeführt. Die Gateway Tests beinhalten nochmals die Sanity Checks und Schwachstellen Tests. Die Tests werden über den API-Gateway ausgeführt, um dessen Funktionsweise zu überprüfen.

3.2.2.1 Autorisierungs-Tests

Basis-Test

Der erste Schritt, der beim Testen einer geschützten API unternommen werden muss, ist zu überprüfen, dass eine Person autorisiert sein muss, um eine gültige Antwort vom Server zu erhalten. Dies wurde getestet, indem eine Anfrage ohne das Anhängen eines Authentification Headers gestellt wurde.

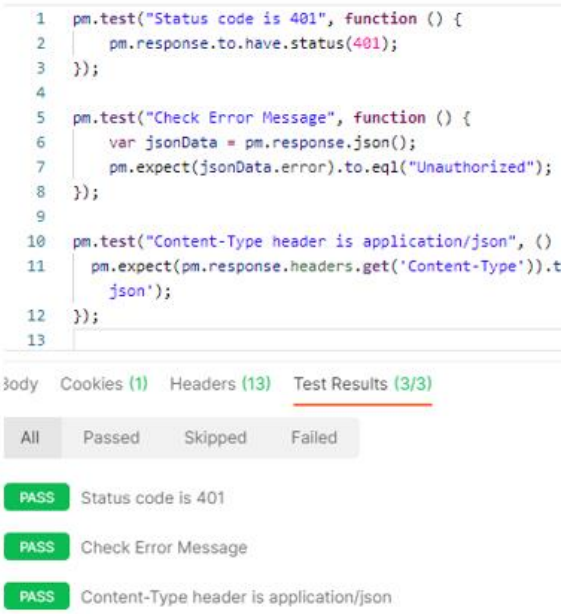


Abbildung 55: Request ohne Authentfication Header

Die entwickelte REST API antwortet bei einer solchen Anfrage immer mit dem HTTP Status Code 401 (Unauthorized).

Broken Function Level Authorization (OWASP Nr.5)

In den Sanity Check Collections wird überprüft, ob Anfragen auf nicht autorisierte Ressourcen korrekt abgelehnt werden. Es wird eine Anfrage mit einem gültigen JWT gestellt. Das JWT hat jedoch nicht den erforderlichen Berechtigungen (Scope) um die entsprechende Aktion durchzuführen

Broken Function Level Authorization (OWASP Nr.5)

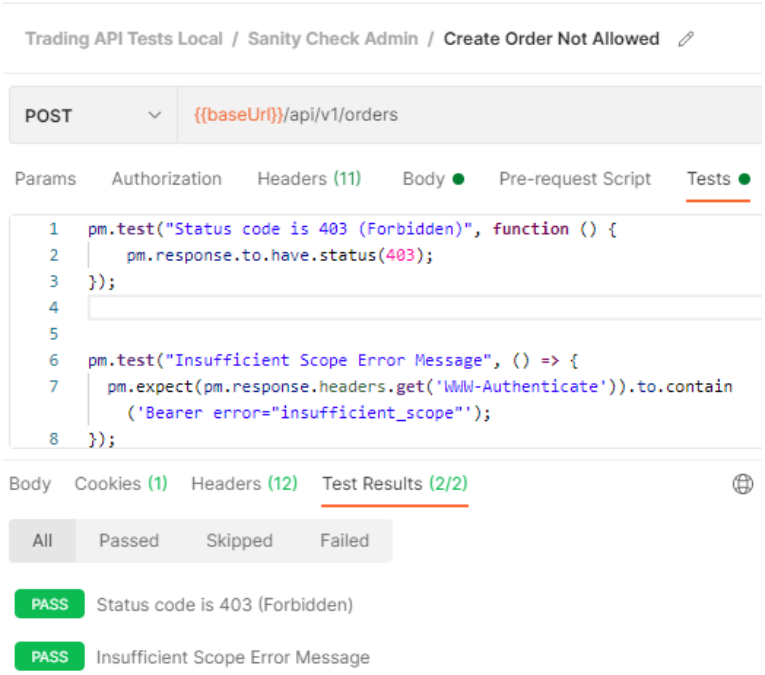


Abbildung 56: POST Request mit ungenügenden Scope

Die Tests überprüfen, ob die API in einem solchen Fall mit dem HTTP Status Code 403 (Forbidden) antwortet. In der obigen Abbildung versucht ein Administrator eine Order zu erstellen. Ein Administrator darf in der Beispiel-API keine Order erstellen, weshalb die API die Anfrage verbietet. Die Error Meldung wird im Response Header zurückgegeben. Der Zugriff darf für die Anfrage in der Abbildung nur erlaubt werden wenn das mitgesendete JWT den Scope “order:create“ besitzt. Mit solchen Tests wird die Schwachstelle Broken Function Level Authorization überprüft, da damit explizit getestet wird, dass der Zugriff auf unberechtigte Ressourcen verboten wird.

Broken Object Level Authorization (OWASP Nr.1)

Um die Schwachstelle Broken Object Level Authorization zu überprüfen wurden Tests erstellt, welche überprüfen, ob ein Benutzer mit einer korrekt ID auf Ressourcen zugreifen kann, welche ihm nicht gehören. Die Tests werden immer dann benötigt, wenn ein Benutzer einen Zugriff auf eine bestimmte Ressource möchte. Ein Kunde darf nicht die Accounts, Positionen und Orders von anderen Kunden sehen, weshalb die nachfolgenden Tests erstellt worden.

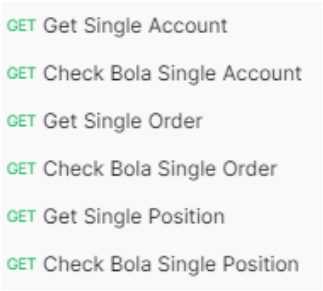


Abbildung 57: BOLA Tests

Die Tests überprüfen, die beschriebene Implementation der Autorisierung in Kapitel 3.1.6. Es wurde definiert, dass die API mit dem HTTP Status Code 404 (Not Found) und einer zugehörigen Fehlermeldung antwortet. Beim nachfolgenden Test versucht ein Kunde auf den Account eines anderen Kunden zuzugreifen. Als Vorbereitung für den Test musste zuerst die Account-ID eines anderen Benutzers gespeichert werden.

Broken Object Level Authorization (OWASP Nr.1)

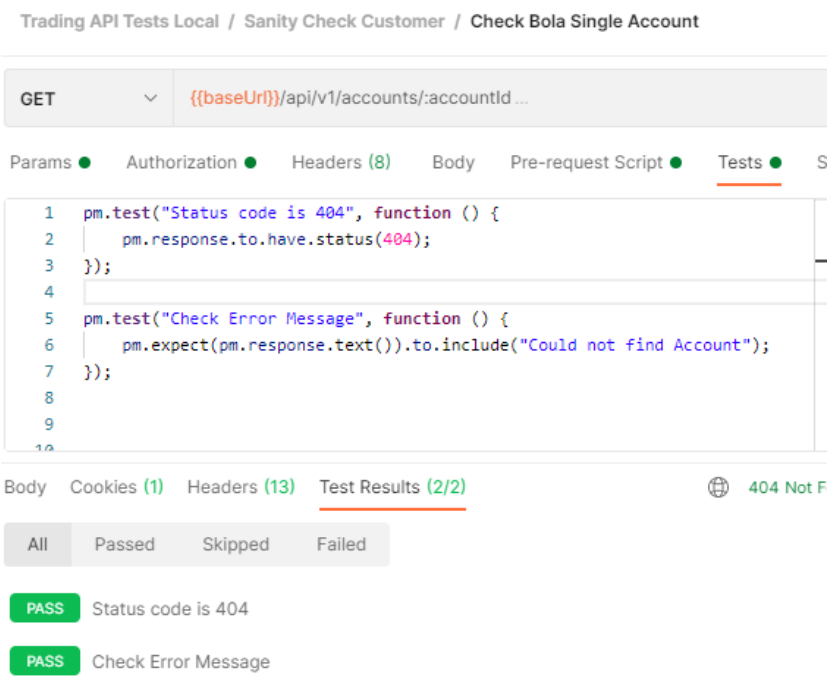


Abbildung 58: Check Broken Object Level Authorization (BOLA)

Der Test überprüft, ob der Zugriff bei diesem Kunden mit dem Status Code 404 abgelehnt wird und Fehlermeldung «Could not find Account» ausgeben wird.

Die genaue gleiche Anfrage wird auch mit dem korrekten Benutzer durchgeführt, um zu verifizieren, dass die Account-ID existiert und die Anfrage erfolgreich beantwortet wird.

3.2.2.2 Schema Validierung Tests

Schema Definition

Um zu verhindern das eine API zu viele Informationen preisgibt (Excessive Data Exposure OWASP Nr.3) wurden für die einzelnen Response-Nachrichten Schema Tests geschrieben. Ein Schema ist die Struktur der API, daher besteht der Zweck dieses Tests darin, zu überprüfen, ob das eingehende Schema mit dem bereits vorhandenen Schema übereinstimmt.

Die nachfolgende Abbildung zeigt exemplarisch das Schema für das Objekt „Order“. Die JSON-Schemas wurden mithilfe der Webseite jsonschema.net erstellt. [57].

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "$id": "https://trading-api-ba21.herokuapp.com/",
    "required": [
        "status",
        "id",
        "type",
        "quantity",
        "price"
    ],
    "properties": {
        "status": {
            "$id": "#/properties/status"
        },
        "id": {
            "$id": "#/properties/id"
        },
        "type": {
            "$id": "#/properties/type"
        },
        "quantity": {
            "$id": "#/properties/quantity"
        },
        "price": {
            "$id": "#/properties/price"
        }
    },
    "additionalProperties": false
}
```

Abbildung 59: JSON Order Schema Beispiel

Unter dem Property „required“ werden die zwingend nötigen Attribute einer Order-Response definiert. Um nicht definierte Attribute zu verhindern, wird „additionalProperties“ auf false gesetzt. Damit wird sichergestellt, dass der Test fehlschlägt, wenn ein neues Order Attribut von einem Entwickler hinzugefügt wird. Die JSON-Daten folgen dieser bestimmten Struktur und können validiert werden, indem sie mit der Antwort auf eine gesendete Anfrage verglichen werden. Bei der Beispiel-API wurden für alle Antworten solche Schema-Checks implementiert.

Tests

Bei der Beispiel-API wurden für alle Antworten solche Schema-Checks implementiert. Es wurde überprüft das der Test fehlschlägt, wenn die Antwort sich verändert und beispielweise neue JSON-Attribute enthält.

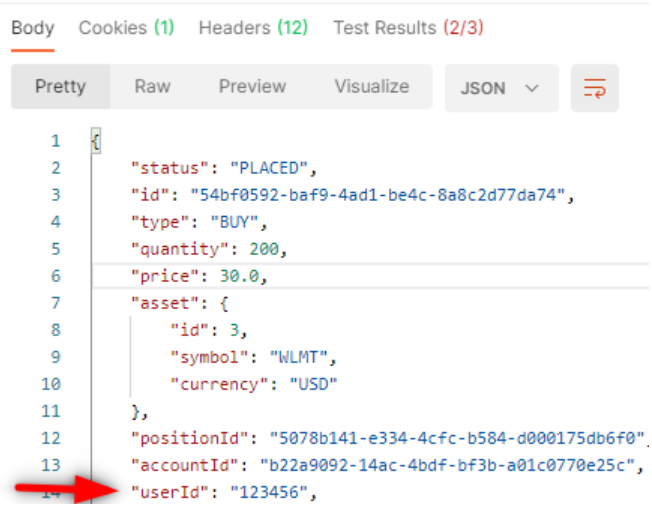


Abbildung 60: Order Response Nachricht mit userId

Die vorgängige Abbildung zeigt, eine Order Response Nachricht, welche neu das zusätzliche Attribut „userId“ beinhaltet. Der Schema-Test erkennt die Änderung und warnt den Entwickler, dass sich die Antwort-Nachricht verändert hat.

Abbildung 61: Test Excessive Data Exposure

Fazit

Mit solchen Tests können zwar Änderungen am Model entdeckt werden, jedoch können damit nicht überprüft werden ob beispielweise ein Array zu viele Objekte beinhaltet. In Zukunft, jedoch nicht im Rahmen dieser Arbeit, könnte auch dafür ein Test erstellt werden

3.2.2.3 Value Tests

Mass Assigment

Um mit einem Test die Schwachstelle Mass Assigment mit Test wurden Value Checks erstellt. Damit eine API nicht anfällig auf diese Schwachstelle ist, ist es entscheidend, dass sie korrekte Werte zurückgibt. Für die Beispiel-API wurde der folgende Endpoint erstellt, welcher die Schwachstelle enthält:

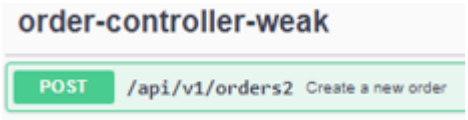


Abbildung 62: Schwachstelle Mass Assigment

Die nachfolgende Abbildung zeigt, das JSON-Objekt, welches von der API erwartet wird um eine Order zu erstellen:

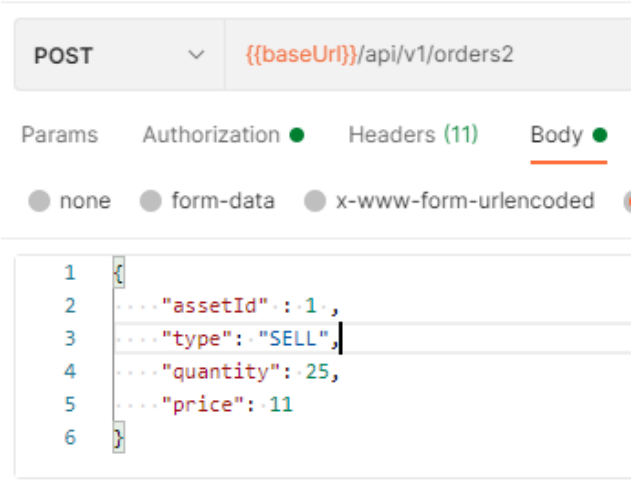


Abbildung 63: Erwartete Eingabe vom User

Erfolgreicher Angriff

Die API erstellt eine Order für den Kunden, welcher sich automatisch im Status „Placed“ befindet. Nur ein Administrator sollte den Status einer Order verändern können, indem sie die Order ausführen oder stornieren.

```
{
    "status": "PLACED",
    "id": "a94f35f3-8df8-46ba-ac6d-db9c43f0f2f3",
    "type": "SELL",
    "quantity": 25,
    "price": 11.0,
    "asset": {
        "id": 1,
        "symbol": "AMZN",
        "currency": "USD"
    },
    "positionId": "813d74f2-aa06-4b22-8569-e0f0d995bd9a",
    "accountId": "88d271b5-4261-4630-a5f4-114cf0d7b84b",
    "_links": {
```


Abbildung 64: Antwort der API

Ein Angreifer könnte nun versuchen den Status der Order bei der Erstellung der Order zu setzen.

Im nachfolgenden Test wird das JSON-Objekt um den Status „Done“ erweitert. Die Antwort zeigt, dass der Order mit dem gewünschten Status erstellt worden ist, obwohl dies vom Entwickler nicht vorgesehen war.

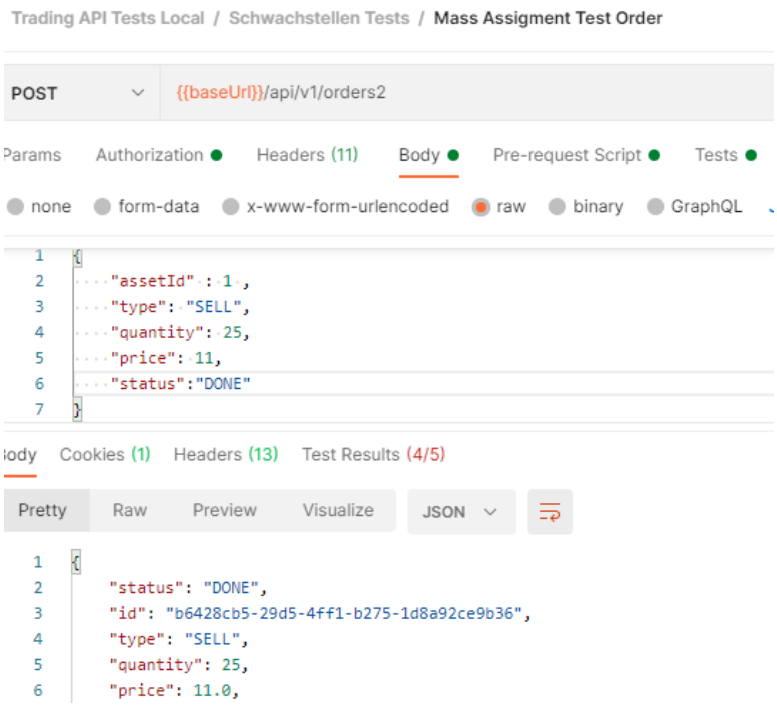


Abbildung 65: Mass Assigment Schwachstelle

Value Checks

Mithilfe von Value Checks können solche Schwachstellen getestet werden. Wenn eine neue Order erstellt wird, muss diese immer den Status „Placed“ haben, wenn nicht schlägt der Test fehl. Beim Schreiben von Tests ist es deshalb wichtig, den Workflow der jeweiligen API zu verstehen.

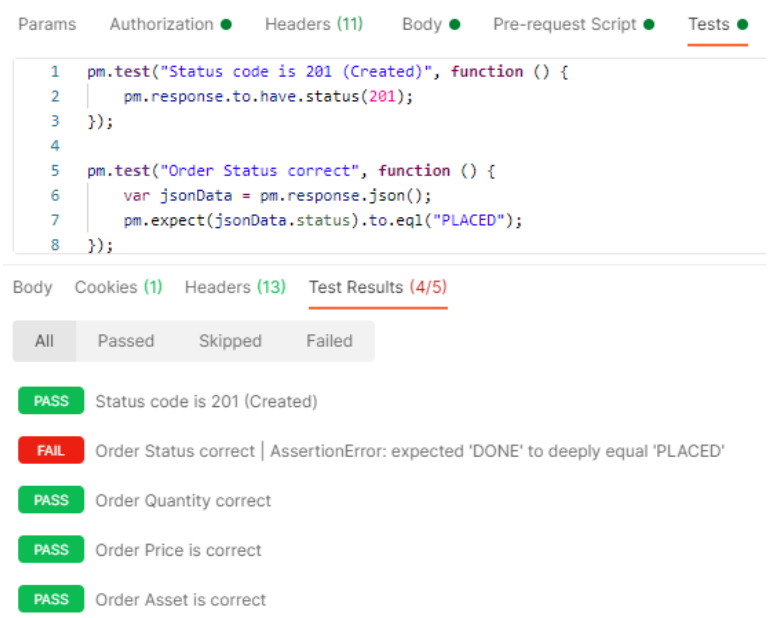


Abbildung 66: Value Checks

Die obige Schwachstelle ist dadurch entstanden, dass bei der Erstellung des Data Transfer Objekts (DTO) ein Default-Wert für das Status-Attribut verwendet wird, welcher vom Benutzer überschrieben werden kann. Weiter wurde in der Mapper-Konfiguration, welche das DTO in eine Entity wandelt, das Status Attribut „Status“ nicht ignoriert und deshalb in die Datenbank gespeichert.

3.2.2.4 Security Header Tests

Security Misconfiguration Tests

Um die Schwachstelle Security Misconfiguration zu testen, wurden die Security Header HSTS und CSP eingesetzt. Der Header HSTS wird lokal nicht gesetzt, da das TLS-Zertifikat erst auf dem Server eingesetzt wird. Für die Überprüfung der Header wurden die nachfolgenden zwei Tests eingebaut.

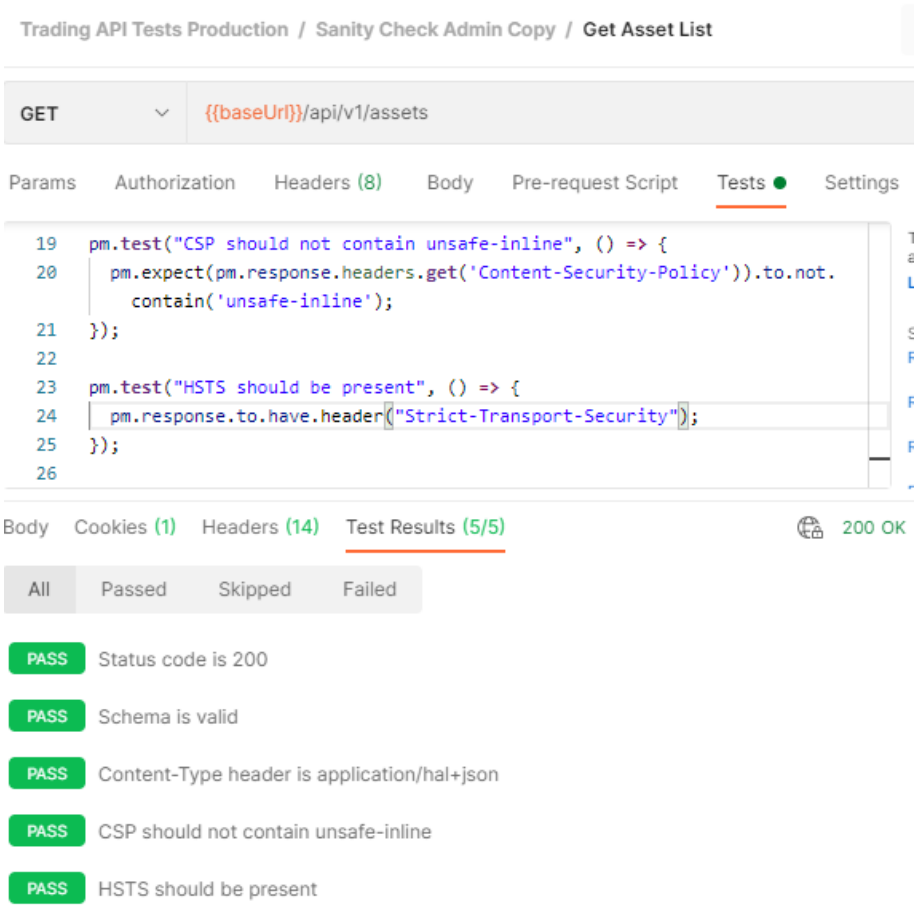


Abbildung 67: Security Header Tests

Der HTTP Strict Transport Security Header (HSTS) muss im laufenden Betrieb immer aktiv sein. Die Content-Security-Policy darf das Keyword „unsafe-inline“ nicht enthalten, da dies die Sicherheit gefährdet.

Die Schwachstelle Security Misconfiguration beinhaltet noch viel mehr als das Überprüfen von korrekten Security Headern. Hierzu gehören auch fehlende Security Patches für eine verwendete Software oder ein System. Hierzu gehören auch Bibliotheken, welche für die Programmierung der Beispielapplikation verwendet wurden. Diese besitzen jeweils selbst eigene Schwachstellen, welche durch ein Code-Review überprüft werden müssen.

3.2.2.5 SQL Injection Tests

Mögliche Wege

Eine API bietet grundsätzlich immer zwei Wege, um Daten von einem Client zu empfangen. Einer ist die Eingabe von Strings im URL Feld und der andere über die Eingabe von Daten, welche im Body einer HTTP Anfrage gesendet werden. welche im Body einer HTTP Anfrage gesendet werden.

Für die Beispielapplikation wurde die nachfolgenden zwei Endpoints erstellt, welche über das URL Feld anfällig sind für eine SQL-Injektion.

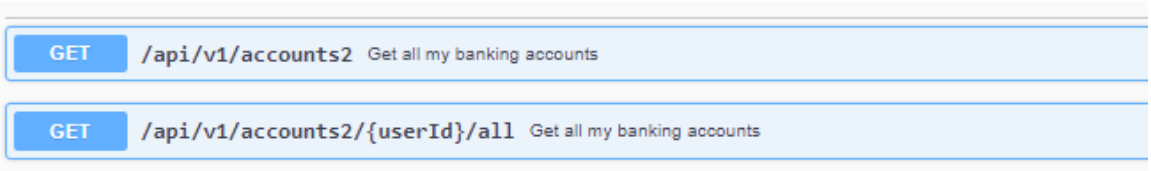


Abbildung 68: SQL-Injection Schwachstellen Endpoints

SQL-Fehlermeldungen

Damit ein Angreifer eine erfolgreiche SQL-Injection durchführen kann, benötigt er Hinweise darauf, dass im Hintergrund eine Datenbankabfrage erfolgt. In der nachfolgenden Anfrage wird anstelle einer 1, eine 1 mit einem Hochkomma in die URL eingefügt. Die API antwortet mit dem HTTP Status Code 500 (Internal Server Error) und gibt die interne Systemfehlermeldung zurück. Eine API sollte niemals solche Fehlermeldungen an den Benutzer leiten. Der Angreifer weiss anhand der Fehlermeldung, dass Hibernate im Hintergrund verwendet wird und sieht das fehlgeschlagen SQL-Statement.

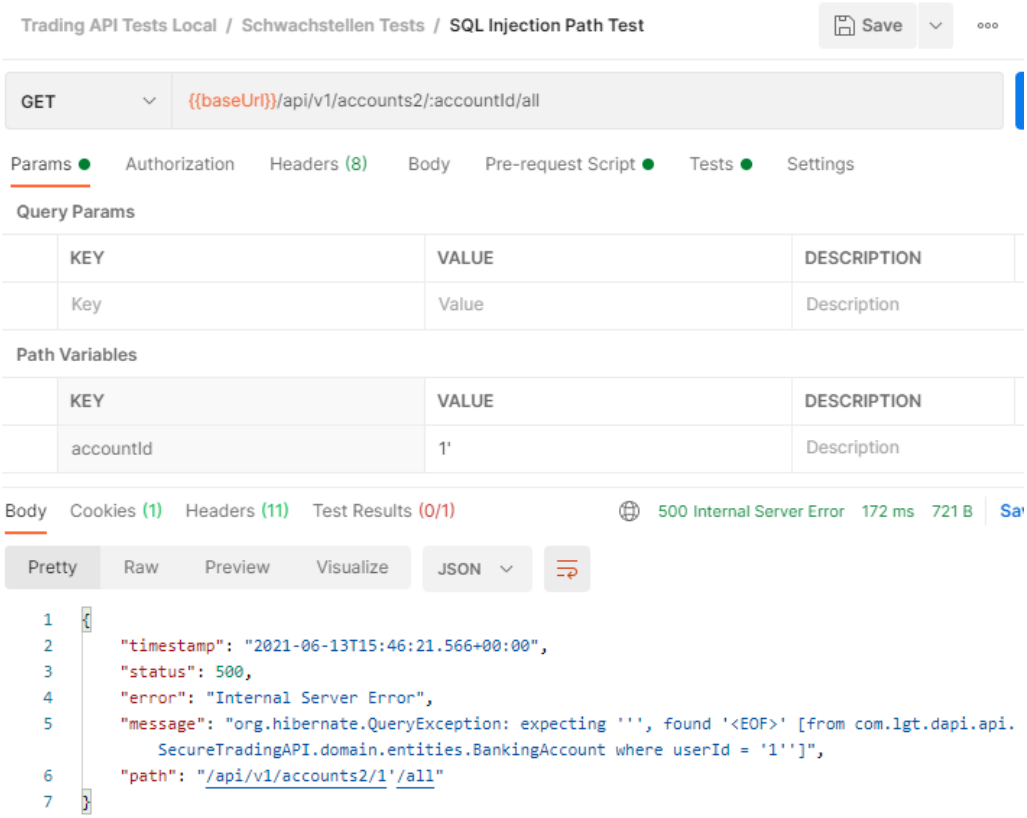


Abbildung 69: SQL-Fehlermeldung

Erfolgreicher Angriff

Aufgrund der obigen Fehlermeldung ist es für einen Angreifer nun leicht eine SQL-Injektion auszuführen, um so beispielweise auf Daten von anderen Kunden zuzugreifen. Die Fehlermeldung zeigt an die Abfrage für die Banking Account Tabelle durchgeführt wird. Damit ein Angreifer alle Accounts der Tabelle erhält muss er die obige Abfrage immer wahr werden lassen. Die nachfolgende Abbildung zeigt die erfolgreiche SQL-Injection mit dem String: *„123‘ or ‚1‘ = ‘1“.* Die API wirft keinen Fehler und retourniert alle Accounts, welche sich in der Datenbank gespeichert hat.

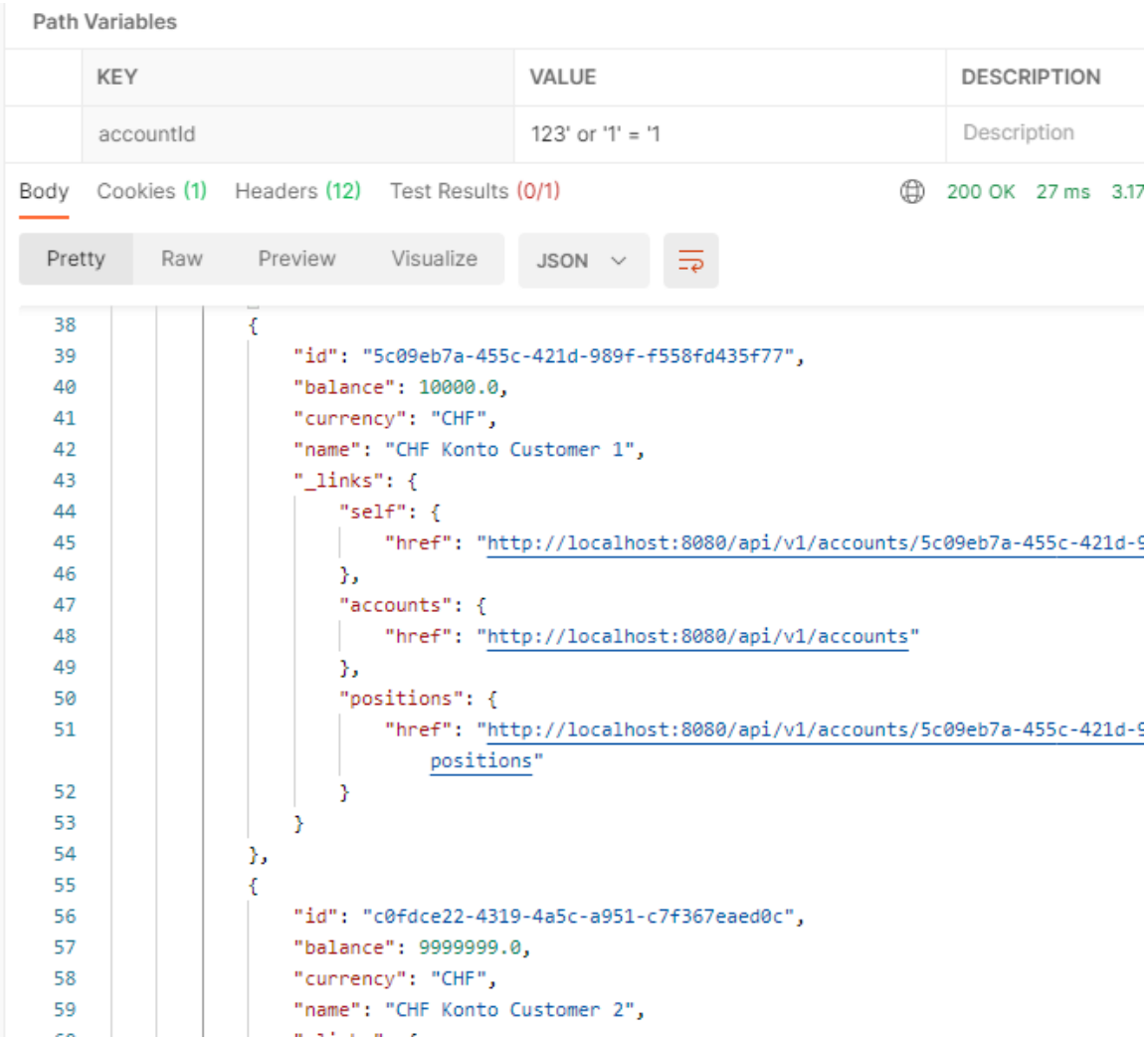


Abbildung 70: Erfolgreicher SQL-Angriff

3.2.2.6 Automatisierung der Test mit Newman

Gitlab CI/CD

Die erstellten Tests werden mit Newman durch die Gitlab CI/CD ausgeführt und in der Pipeline angezeigt.

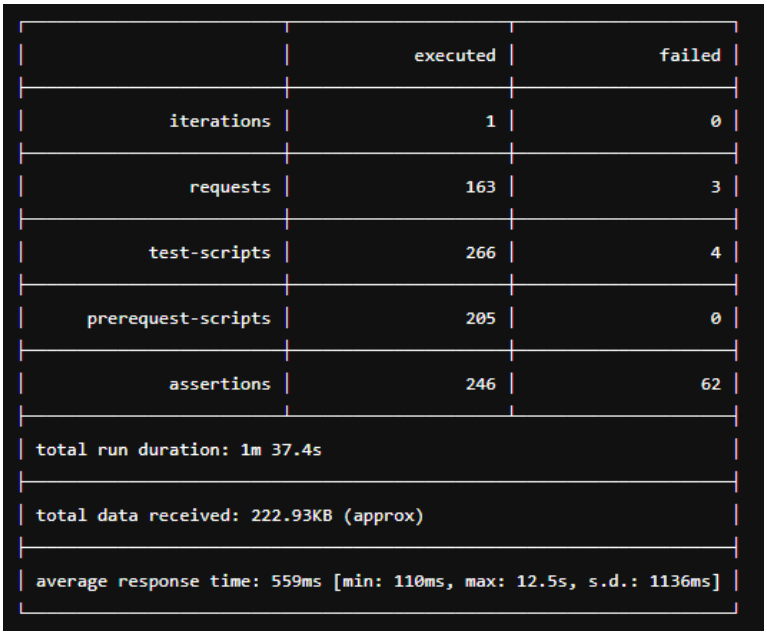


Abbildung 71: Gitlab CI/CD Newman Pipeline

HTML Report

Ebenfalls wurde mit Newman ein HTML Report erstellt, welcher die Resultate der Tests zusammenfasst. Die nachfolgende Abbildung zeigt das Dashboard der Tests. Es wurden insgesamt 163 Request durchgeführt und 247 Tests waren erfolgreich. Die Tests welche die Endpoints, mit den eingebauten Schwachstellen testen schlagen alle fehl. Der Report ist auf dem Gitlab Repository der Ost verfügbar im BA-Projektordner gibt es einen Unterordner newman, wo der Report betrachtet werden kann.

Newman Dashboard

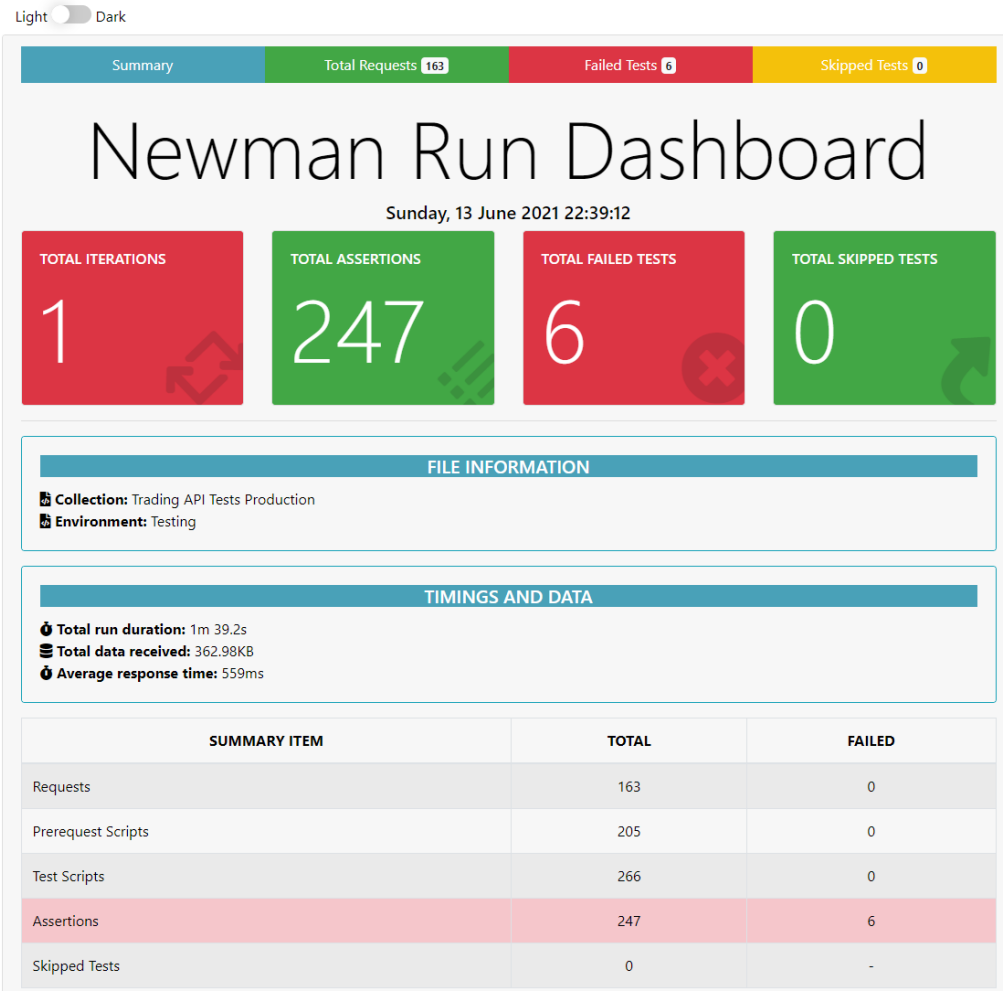


Abbildung 72: Newman Dashboard

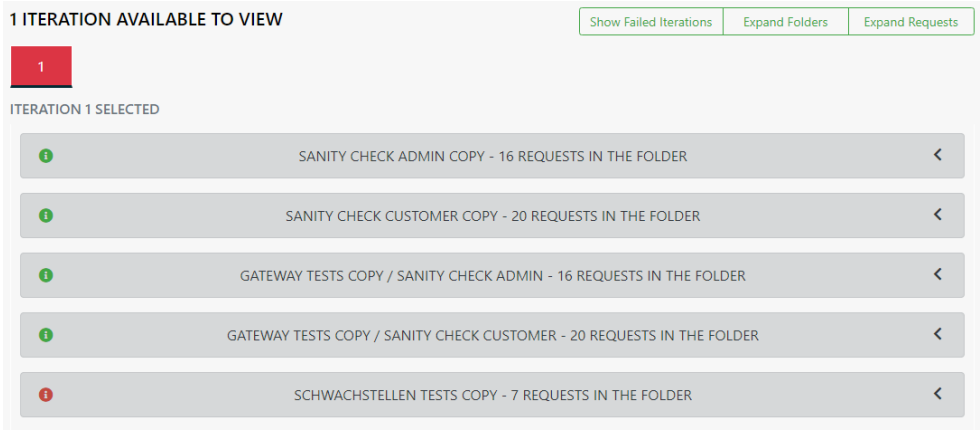


Abbildung 73: Anfragen gruppiert nach Ordner

Detail Ansicht

Der erstellte HTML Report bietet auch die Möglichkeit einzelne Requests, welche für einen Test ausgeführt wurden genauer zu betrachten. Die Header und Body Inhalte sind für jeden Request und jede Response verfügbar. Ausserdem werden auch die ausgeführten Tests angezeigt.

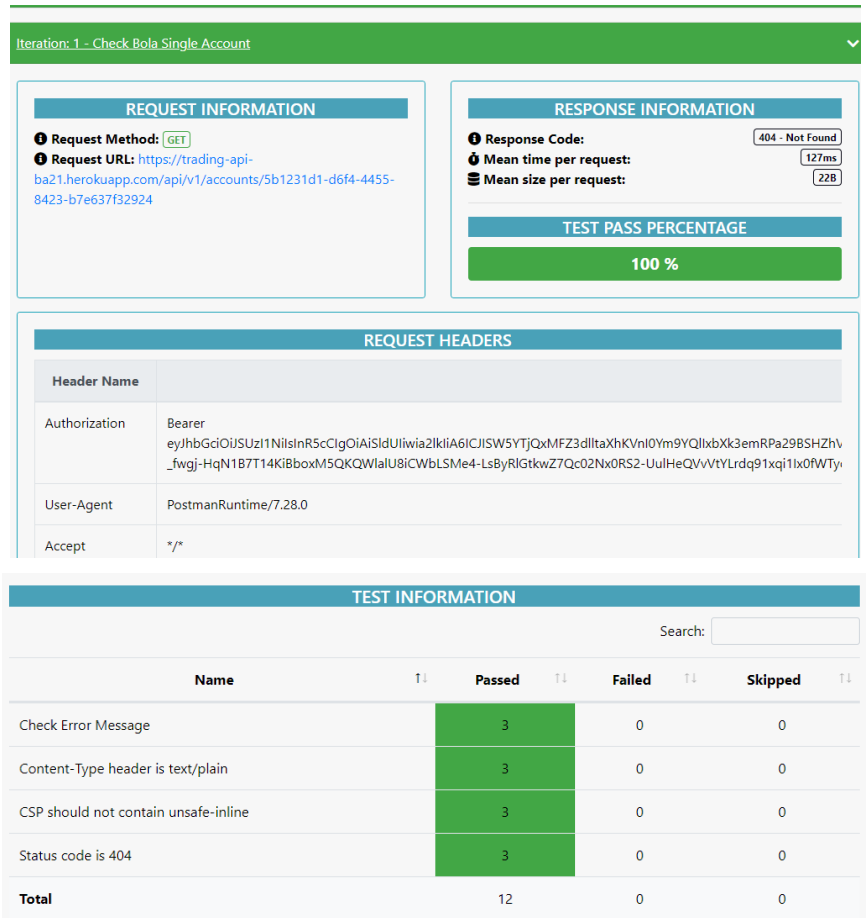


Abbildung 74: Ansicht eines einzelnen Tests

3.2.3 Statisches Code Analyse

Snyk

Für die statische Code Analyse wird die Software Snyk genutzt. Snyk bietet Sicherheit für Entwickler mit automatischer Erkennung und Behebung von Schwachstellen. Ihre führende Schwachstellendatenbank ermöglicht eine umfassende Open-Source-Abdeckung und geht über CVE-Schwachstellen und andere öffentliche Datenbanken hinaus. Snyk bietet ausserdem enge Integrationen mit allen führenden Container- und CI/CD-Plattformen.

DevSecOps

DevSecOps bedeutet, von Anfang an über Anwendungs- und Infrastruktursicherheit nachzudenken und fordert von den Entwicklern mehr Verantwortung für die Sicherheit zu übernehmen. Damit Entwickler mehr Verantwortung für die Sicherheit übernehmen können, müssen sie in der Lage sein, die Sicherheit so früh wie möglich im Lebenszyklus der Softwareentwicklung und auf möglichst einfache Weise in ihren Entwicklungsworkflow zu integrieren. Genau hier setzten statische Code Analyse Tools wie Snyk an.

Verwendung von CVSS

Snyk verwendet zur Bewertung der gefunden Schwachstellen das Common Vulnerability Scoring System (auch bekannt als CVSS-Scores) und stuft die Wichtigkeit der Schwachstellen nach dem CVSS Score ein:

Wichtigkeit	CVSS v3 Rating
Kritisch	9.0 – 10.0
Hoch	7.0 – 8.9
Medium	4.0 – 6.9
Low	0.1 – 3.9

Verwendung

Die Verwendung von Snyk in der CI/CD ist kostenpflichtig, da ein API Key benötigt wird. Snyk wurde deshalb bei dieser Arbeit nicht in den CI/CD Pipeline integriert. Der Einsatz eines Tools wie Snyk sollte schon sehr früh im Softwareentwicklungs-Lebenszyklus (SDLC) stattfinden, da es keine funktionierende Applikation benötigt und ohne die Ausführung von Code erfolgen kann. Für die Beispiel-API wurde das im April 2021 veröffentlichte JetBrains DIE-Plugin von Snyk verwendet. Mit dem neuen Plugin können sich Entwickler während der Entwicklung ihrer Applikation mit der Sicherheit ihrer gesamten Codebasis befassen, ohne ihren Arbeitsablauf zu unterbrechen, und letztlich schneller sicheren Code ausliefern. (vgl. [58]).

Die nachfolgende Abbildung zeigt die gefunden Schwachstellen der Beispiel-API:

Gefunde Schwachstellen

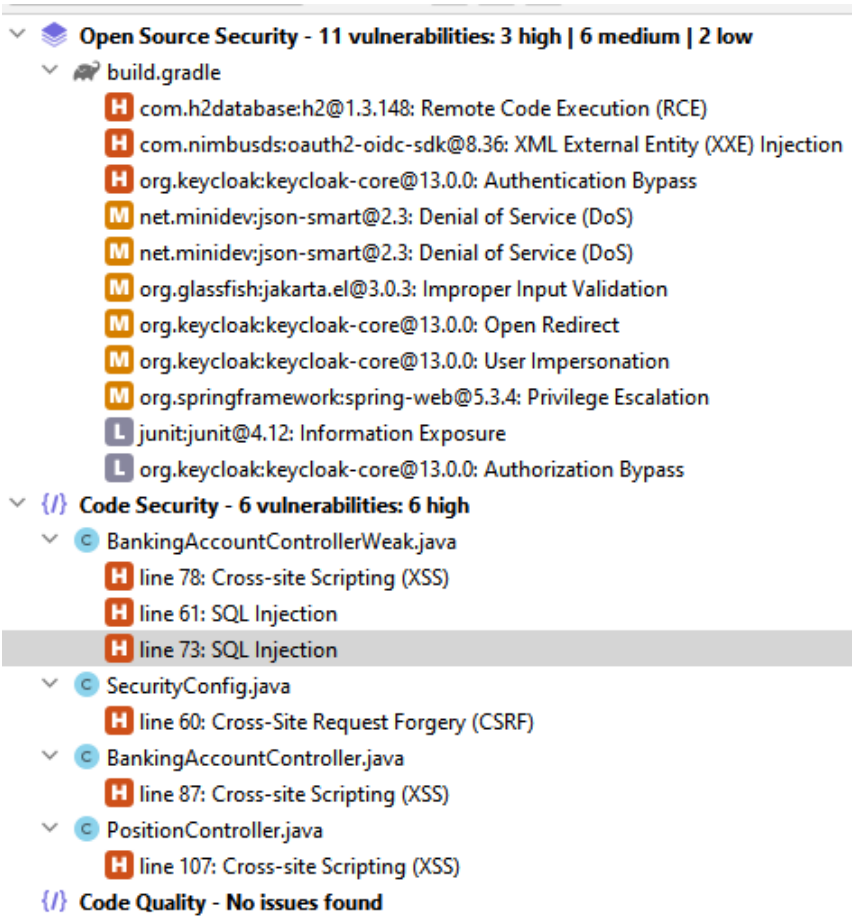


Abbildung 75: Entdeckte Schwachstellen mit Snyk

Die Verwendung des Plugins ist sehr einfach und intutiv. In wenigen Sekunden liefert das Plugin eine von Schwachstellen und Problemen, die in drei Kategorien eingeteilt sind:

- **Open-Source-Sicherheit** sind bekannte Schwachstellen in den direkten und indirekten Abhängigkeiten von Open-Source-Bibliotheken im Projekt.
- Unter **Code-Sicherheit** werden identifizierte Sicherheitsschwachstellen im eigenen Code angezeigt
- Die **Codequalität** zeigt Probleme mit der Codequalität des eigenen Codes.

In der obigen Abbildung ist ersichtlich das keine Probleme mit der Code-Qualität bestehen. Unter Code-Sicherheit wurden die eingebauten SQL-Schwachstellen entdeckt und weitere Cross-Site Scripting (XSS) Schwachstellen. Die meisten Schwachstellen wurden in den extern verwendeten Bibliotheken der Applikation gefunden.

Resultat Code Security

Im Rahmen dieser Arbeit gehen wir nur auf die Schwachstellen, welche mit Hoher Priorität bewertet wurden, ein. Unter Code Security wurden die bewusst erstellten SQL-Schwachstellen erkannt und mit Hoher Priorität bewertet:

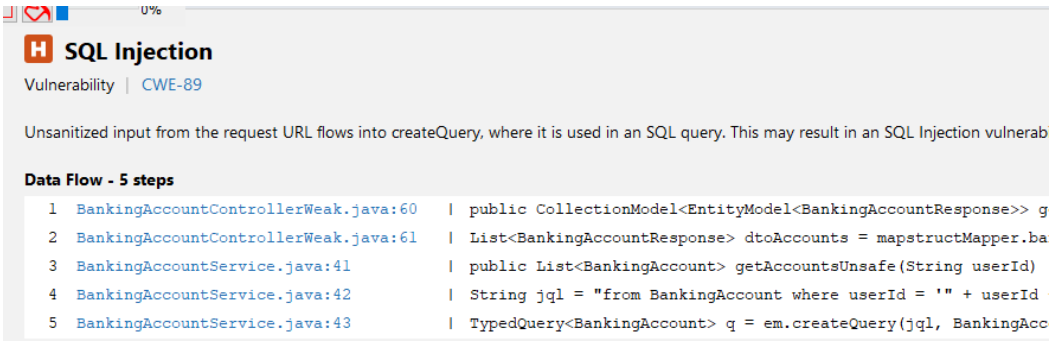


Abbildung 76: Entdeckte SQL-Schwachstelle

Neben der SQL-Schwachstelle wurden auch diverse XSS-Schwachstellen gefunden. Es handelt sich immer um die gleiche Schwachstelle, welche auf mehreren Endpoints vorhanden ist. Snyk erkennt bei den jeweiligen Endpoints, dass diese Parameter entgegennehmen und diese in der Antwort reflektieren, ohne ein Input Sanitization durchzuführen. Die API selbst rendert zwar keine HTML Seite, wenn jedoch eine SPA-Applikation die API verwendet und die Daten der API ebenfalls nicht überprüft, ist eine erfolgreiche Reflected-XSS Attacke möglich.

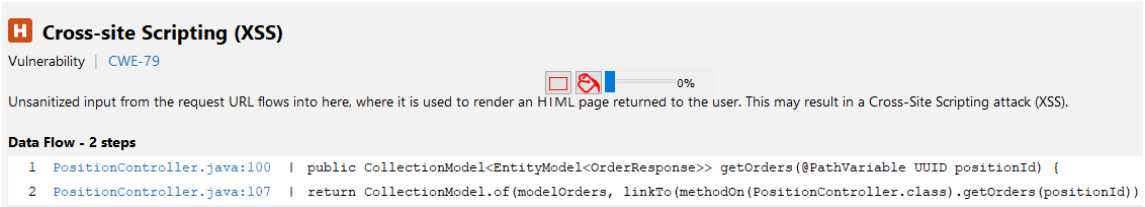


Abbildung 77: XSS Schwachstelle

Bei der Security Konfiguration wurde manuell der CSRF-Header deaktiviert, weshalb das Snyk Tool hier eine Warnung anzeigt. Der CSRF-Header ist bei der API aber nicht nötig, da bei jeder Anfrage an die API der „Authorization Bearer“ Header verwendet wird. Diese Authentifizierungsmechanismus schützt vor CSRF Angriffen. Ein Angreifer kann keine Browser-Anfrage mit einem Authorization Header und dem korrekten Token an ein Opfer senden. Dies hängt damit zusammen das für den API-Aufruf ein Authorization-Header anhängt werden muss, so kann sich ein Browser nicht automatisch identifizieren und CSRF ist nicht möglich (vgl. [58]). Die Warnung kann mit der aktuellen Authentifizierung demzufolge ignoriert werden.

Abbildung 78: CSRF Schwachstelle

Snyk Reporting

Mit dem Befehl snyk monitor wird ein Projekt auf der Snyk-Website erstellt, das kontinuierlich auf neue Sicherheitslücken überwacht wird. Die Schwachstellen der verwendeten Bibliotheken werden im Rahmen dieser Arbeit nicht analysiert. Zur

Veranschaulichung zeigt die nachfolgende Abbildung die gefunde H2-Datenbank Schwachstelle:

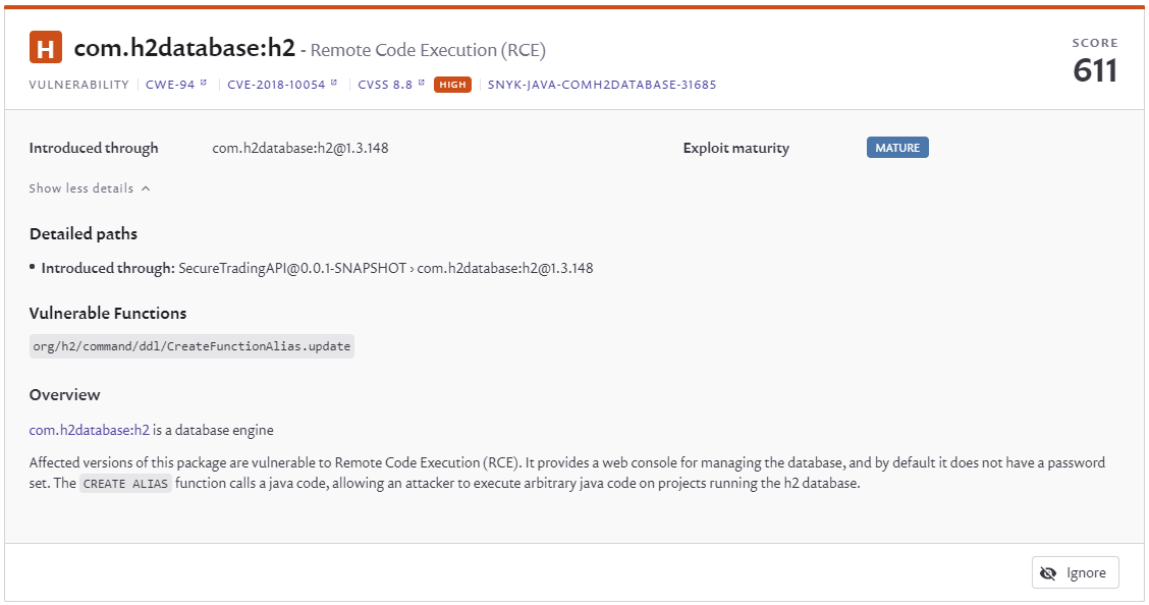


Abbildung 79: Schwachstelle H2 Datenbank

Die verwendete H2 Datenbank für Integration-Tests ist bekannt für eine Remote Code Execution (RCE), weil sie eine Web-Console für die Datenbank anbietet und standardmässig kein Passwort hat. Für die eingesetzte H2 Datenbank wurde ein Passwort verwendet, um die Schwachstelle zu mindern. Die Schwachstelle wird mit Hoher Priorität bewertet da sie einen sehr hohen CVSS Score von 8.8 besitzt. Es empfiehlt sich deshalb dringend einen anderen Datenbank-Typ für Integration-Tests zu verwenden.

Fazit

Eine bemerkenswerte Sache in Bezug auf diese Warnung ist, dass Snyk vollständige Details zu jedem gefundenen Problem liefert. Es enthält Angriff, Beweise, Beschreibung, andere Informationen und Verweise. Es schlägt auch die Lösung der Probleme vor. Dies kann Entwicklern wirklich helfen, ihre Anwendung sicherer zu machen.

3.2.4 Dynamische Sicherheitstests

Einleitung

Für die dynamischen Applikation Sicherheitstests (DAST) wurde der Zed Attack Proxy (ZAP) von OWASP verwendet. Die OWASP Foundation ist eine Non-Profit-Organisation und bietet kostenlos den Zed Attack Proxy an. Es handelt sich um ein Tool, das zum Auffinden von Schwachstellen in Webanwendungen verwendet wird. OWASP definiert ZAP als ein einfach zu bedienendes integriertes Tool für Penetrationstests von Webanwendungen, um Schwachstellen zu finden (vgl. [59]). Das Tool ist bekannt und wird von Anfängern und professionellen Penetrationstestern verwendet werden kann. Es ist optimal für automatisierte Sicherheitstests.

Interception-Proxy

ZAP ist ein sogenannter Interception-Proxy. Das bedeutet, dass alle Anfragen des Benutzers an die Applikation und alle Antworten der Applikation an den Benutzer durch ZAP gesehen werden können. Er fungiert als "Man-in-the-Middle" zwischen dem Client und der der entsprechenden Applikation. Mit ihm können alle HTTP/S Nachrichten abgefangen und modifiziert werden.

API Testing

ZAP wurde eigentlich für das Testen von Webapplikationen entwickelt. Er versteht aber API-Formate wie JSON und XML und kann deshalb auch für das Testen von APIs verwendet werden. Das Problem ist meist, wie man die Endpoints der API effektiv erkunden kann (vgl. [60]). Um die Beispielapplikation zu testen wurden die End-to-End Security Tests von Postman über ZAP geleitet. Dazu musste bei den Postman-Einstellungen der Proxy eingestellt werden.

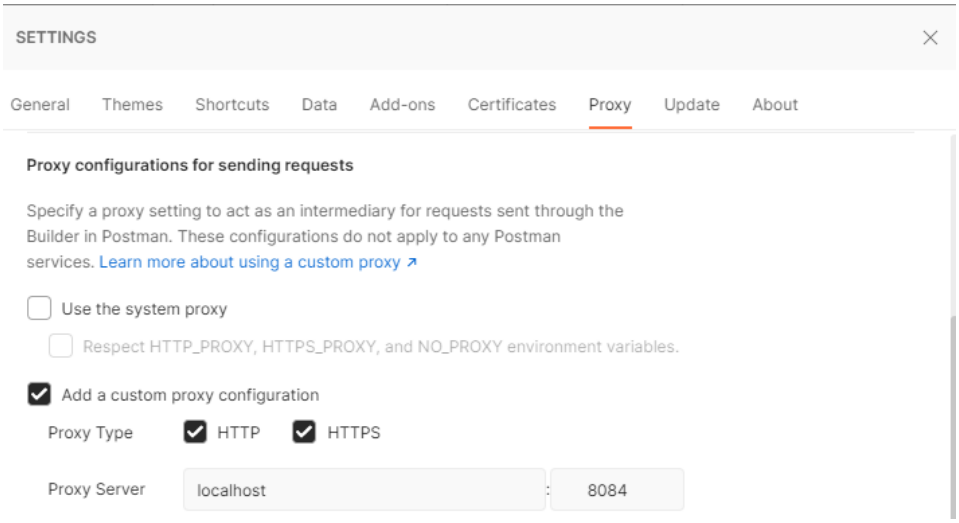


Abbildung 80: Postman Proxy Settings

Aktiver Scanner und Passiver Scanner

Der ZED Attack Proxy unterstützt Aktive Scanner und Passive Scanner. Aktive Scanner greifen aktiv in die API ein, um Schwachstellen zu finden, während passive Scanner nur die Antworten der Applikation an den Browser scannen. Aktives Scannen ist riskanter, da es der Applikation Schaden zufügen kann. Daher darf man einen Aktiven Scanner nicht ohne die Erlaubnis des Eigentümers der Applikation verwenden.

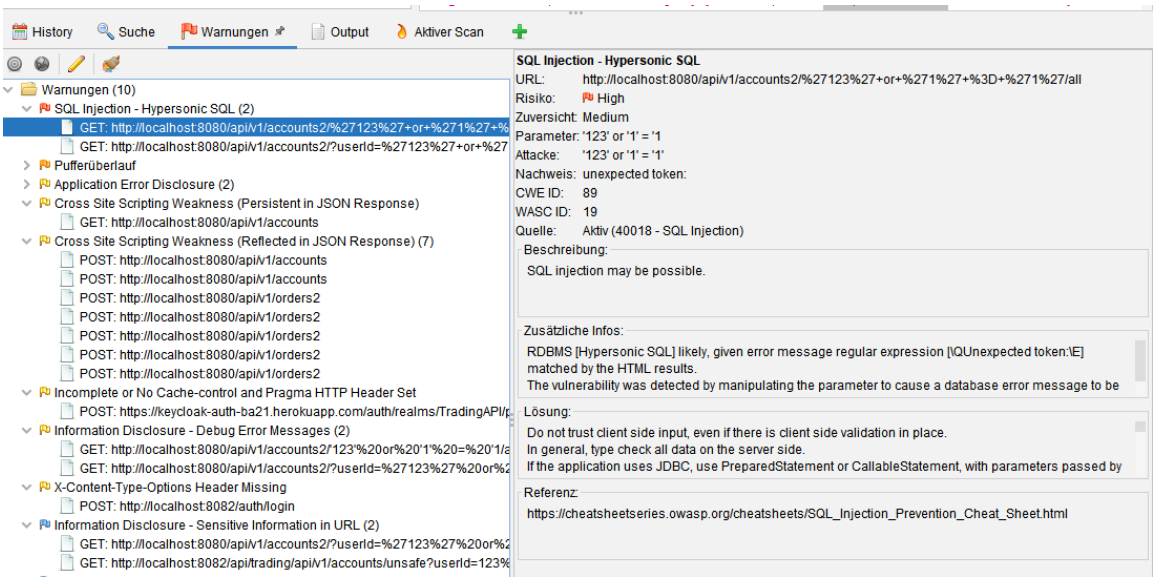
Der Passive Scanner ist sicher in der Anwendung, da er die von der Anwendung erhaltenen Antworten nicht verändert.

Penetration Test

Um einen automasierten Penetrationtest mit ZAP für die API durchzuführen wurden die jeweiligen Anfragen der Postman Tests, welche alle möglichen Routen der API besuchen und testen, über ZAP geleitet. Nachdem in Postman der Proxy entsprechend eingestellt

wurde und alle Tests durchlaufen wurde kann in ZAP über den Kontextmenüeintrag Angriff ein Aktiver Scan gestartet werden. Am Ende des Scans zeigt ZAP die Ergebnisse zusammengefasst und gruppiert in verschiedenen Kategorien von Alerts an. Bei den Alerts handelt es sich um potenzielle Schwachstellen, die in die Kategorien hohe Priorität, mittlere Priorität, niedrige Priorität und reine Information eingeteilt wurden. Dadurch soll das Schadensausmass des damit verbunden Risikos angeben werden. Die Alarmkategorien werden durch unterschiedliche Farbkennzeichnungen angezeigt.

Resultat



Mit einem aktiven Scan können nur grundlegende Probleme gefunden werden. Die SQL Injections und XSS-Schwachstellen, welche bereits mit der statischen Code Analyse entdeckt worden sind, wurden auch vom Aktiven Scan gefunden. Zum Beispiel können die logischen Schwachstellen in der Autorisierung nicht durch einen aktiven Scan gefunden werden.

Wie Snyk bietet auch ZAP vollständige Details zu jedem gefunden Problem vor. Man kann die Alert-Details über den Dialog "Edit Alert" bearbeiten, indem man auf einen Alert doppelklickt, ihn in einem html- oder xml-Format speichert und ihn auch mit einer anderen Sitzung über das Register "Report" im Hauptmenü vergleicht

4 Schlussfolgerungen

Inhalt Das Projekt wurde nach 360 Arbeitsstunden, verteilt über 17 Wochen, abgeschlossen. Die folgenden Kapitel bewerten die erreichten Resultate und geben eine Schlussfolgerung zum Projekt

4.1 Bewertung der Ergebnisse

Inhalt Das folgende Kapitel bewertet die Zielerreichung der einzelnen Ziele. Alle Ziele konnten erfolgreich umgesetzt werden

4.1.1 Analysen

Tool-Evaluation für End-to-End Security Tests Zu Beginn der Arbeit wurde eine Tool-Evaluation über aktuelle Tools zum Testen einer API durchgeführt. Durch die Evaluation wurde ein Tool bestimmt werden, mit welchem die End-to-End Tests an der API durchgeführt werden können. In der Evaluation wurde drei verschiedene Testing-Tools miteinander verglichen und bewertet. Es hat sich gezeigt, dass die Tools sich in einigen Kriterien unterscheiden. Anhand der gesammelten Erkenntnisse wurde mit Praxispartner zusammen entschieden, dass die Tests mit dem Postman Tool durchgeführt werden.

Analyse API Security Grundlagen Um zu verstehen was API Security Testing überhaupt bedeutet, wurde untersucht welche Angriffe auf APIs erfolgen und weshalb sich ihr Risikoprofil von einer Webapplikation unterscheidet. Die gesammelten Erkenntnisse sind bei der Erstellung des Lösungskonzept miteingeflossen.

Lösungskonzept entwickeln Es wurde ein Konzept entworfen, um die Sicherheit der Beispiel-API zu testen. Im ersten Schritt wurden Ziele und Sicherheitsanforderungen für die API definiert. Um die definierten Sicherheitsanforderungen erfüllen zu können wurde in Absprache mit dem Praxispartner darauf aufbauend eine Architektur erstellt. Es wurde entschieden einen globalen Service für die Authentifizierung und einen API Gateway einzusetzen. Da nicht alle Schwachstellen des Systems im Rahmen der Arbeit getestet werden können, wurde eine Bedrohungsmodellierung für die Architektur durchgeführt. Damit konnten zu testende Schwachstellen identifiziert und priorisiert werden.

4.1.2 Entwicklung

Spring REST API Die REST API-Schnittstelle wurde entsprechend den Use-Cases aus der Anforderungsspezifikation umgesetzt und verwendet den Content Type application/hal + json Alle Use-Cases konnten erfolgreich umgesetzt werden und es wurden zusätzliche Schwachstellen für das Testing eingebaut. Für die API wurde eine OpenAPI und RestDocs Dokumentation erstellt.

Datenbank Das Datenbankschema, wurde nach dem Code-First Ansatz umgesetzt. Für die Integration Tests wurde eine H2 In-Memory Datenbank verwendet und für den Betrieb eine PostgreSQL Datenbank.

Authentifizierung & Autorisierung Die Authentifizierung & Autorisierung wurde mit der Keycloak Anwendung umgesetzt und verwendet ein sogenanntes standarisiertes JSON Web Access Token, kurz JWT. Es wurde ein API Gateway aufgesetzt und so konfiguriert, dass alle Anfragen von Benutzern

darüber geleitet wird. Er beschränkt die Anzahl der Aufrufe und zeichnet sie auf. Für die Authentifizierung bildet er aus der Signatur des JWT-Tokens einen Opaque-Token. Die Header und Payload Informationen des JWTs bleiben dadurch dem Benutzer verborgen und sind nur dem API Gateway bekannt.

Continuous Integration und Continuous Development

Die Gitlab CI/CD Pipeline erstellt die Spring Boot API automatisch. Vor dem Build werden Unittests und Integrationstests für die Applikation ausgeführt. Nur wenn diese fehlerfrei ausgeführt wurden, wird das JAR-File der Applikation erstellt. Die Testabdeckung wird in Prozent mit dem Tool Jacoco ermittelt und in der Gitlab Pipeline nach jedem Commit angezeigt. Die API wird anschliessend automatisch auf die Heroku Cloud Plattform ausgeliefert und installiert. Die über das Web verfügbare API wird im letzten Schritt durch die geschrieben Security Tests mit Postman überprüft. Dazu wird die erstellte Postman Collection von der Gitlab CI Pipeline ausgeführt und eine Zusammenfassung in Form eines HTML Reports erstellt.

4.1.3 Testing

Unit- und Integration-Tests

Bei jedem Commit werden die Unit- und Integrationstests automatisiert ausgeführt. Die CI/CD-Pipeline in Gitlab schlägt fehl, wenn die Tests fehlschlagen. Die Testabdeckung wird berechnet und in der Pipeline ausgebeben.

End-to-End Security Tests

Alle priorisierten Sicherheitstests konnten wie geplant implementiert werden und werden automatisch nach jedem Deployment automatisch durch die CI/CD-Pipeline ausgeführt. Die nachfolgenden Schwachstellen wurden mit Priorität 1 bewertet und mithilfe von Tests überprüft:

- Broken Object Level Autorisation (OWASP Nr 1)
- Excessive Data Exposure (OWASP Nr 3)
- Broken Function Level Authorization (OWASP Nr 5)
- Mass Assigment (OWASP Nr 6)
- Security Misconfiguration (OWASP Nr 7)
- Injection (OWASP Nr 8)

Autorisierungstests

Mit Autorisierungs-Tests konnten die beiden Schwachstellen Broken Object Level Authorization und Broken Function Level Authorization getestet werden. Die Schwachstellen können leicht überprüft werden und sollten bei jeder API zwingend getestet werden.

Schema Validierungs-Tests

Mit Schema Validierungs-Tests konnte gezeigt werden, wie die Schwachstelle Excessive Data Exposure getestet werden kann. Der Schema Test schlägt fehl, sobald sich die Antwort-Nachricht der API in ihrer Struktur ändert. Solche Schema Tests sind für jede API empfehlenswert, sie können aber nicht erkennen, wenn beispielsweise eine zu grosse Liste von Orders an den Benutzer ausgeliefert werden. Solche Probleme können nur durch einen Menschen bei einem manuellen Test entdeckt werden.

Value Tests

Um mit einem Test die Schwachstelle Mass Assigment mit Test wurden Value Checks erstellt. Damit eine API nicht anfällig auf diese Schwachstelle ist, ist es entscheidend, dass sie korrekte Werte zurückgibt.

Security Header Tests

Um die Schwachstelle Security Misconfiguration zu testen, wurden die Security Header HSTS und CSP eingesetzt. Der Header HSTS wird lokal nicht gesetzt, da das TLS-Zertifikat erst auf dem Server eingesetzt wird.

SQL Injection Tests

Die eingebauten SQL-Schwachstellen konnten erfolgreich getestet werden, da sie SQL-Fehlermeldungen an den User zurückliefern und den HTTP Status Code 500 (Internal Error

Statische Code Analyse

Es wurde eine statische Code Analyse mit dem CLI-Tool Snyk durchgeführt. Jede Durchführung wird an das Snyk Dashboard gesendet und kann dort analysiert werden. Der Einbau in die Gitlab CI/CD ist nicht erfolgt da dies kostenpflichtig wäre.

Snyk konnte die eingebauten SQL-Schwachstellen und zusätzliche Cross-Site Scripting (XSS) Schwachstellen entdecken, welche nicht bewusst in die Applikation eingebaut worden sind. Ausserdem konnten diverse Schwachstellen mit einem hohen CVSS Score in den verwendeten externen Bibliotheken ausfindig gemacht werden. Der Einsatz von einem statischen Code Analysetool wie Snyk wird deshalb dringend empfohlen. Eine bemerkenswerte Sache in Bezug auf die Warnung ist, dass Snyk vollständige Details zu jedem gefundenen Problem liefert. Es enthält Angriff, Beweise, Beschreibung, andere Informationen und Verweise. Es schlägt auch die Lösung der Probleme vor. Dies kann Entwicklern wirklich helfen, ihre Anwendung sicherer zu machen.

Penetration Test

Es wurde ein automatisierter Pentest mit dem DAST-Tool ZAP durchgeführt. Dazu wurde ein Aktiver Scan durchgeführt. Mit einem aktiven Scan können nur grundlegende Probleme gefunden werden. Die SQL Injections und XSS-Schwachstellen, welche bereits mit der statischen Code Analyse entdeckt worden sind, wurden auch vom Aktiven Scan gefunden. Zum Beispiel können aber die logischen Schwachstellen in der Autorisierung nicht durch einen aktiven Scan oder Snyk gefunden werden.

4.2 Fazit

Die Bewertung der Ergebnisse zeigt, dass die geplanten Ziele vollumfänglich umgesetzt werden konnten. Am Anfang war nicht gänzlich klar, ob die Architektur wirklich nach Plan implementiert, werden können, da aufgrund von fehlender Erfahrung eine Aufwandschätzung schwierig war. Es ist deshalb besonders erfreulich, dass die Architektur wie geplant umgesetzt werden konnte. Die Tool-Evaluation hat mehr Zeit in Anspruch genommen als geplant, weil bei der Umsetzung der Testfälle einige Hürden aufgetaucht sind. Zu Beginn war auch unklar, ob der API-Gateway die Autorisierung durchführen soll oder die API selbst, da für letzteres mehr Aufwand betrieben werden muss. Der Entwickler muss sich zusätzlich um die Zugriffskontrolle kümmern. Es hat sich jedoch schnell gezeigt, dass die Umsetzung solcher Zugriffskontrollen mit Spring Security recht simpel ist und Vorteile in Bezug auf die korrekte Validierung eines Access Tokens bietet.

4.3 Reflexion der Arbeit

Die Umsetzung der Bachelorarbeit hat mir viel Freude bereitet. Ich habe durch die Bachelorarbeit viele neue Themenbereiche kennengelernt und konnte dadurch sehr viel Neues dazulernen. Die Entwicklung einer API mit Spring Boot hat mich begeistert und ich freue mich auf die Entwicklung weiterer APIs. Die Arbeit allein war teilweise schwierig, da es viele verschiedene Aspekte der Sicherheit gibt und man sich darin schnell verläuft. Der OAuth2 Standard beispielweise ist hochspannend, aber auch sehr umfangreich. Die wöchentlichen Meetings mit meiner Betreuerin waren hilfreich und motivierend. Frau Weiler konnte mir gute Inputs für die Arbeit geben. Ich habe recht viel Zeit in die Authentifizierung und Autorisierung gesteckt, da mich das Thema besonders interessiert hat.

4.4 Ausblick

Es konnte gezeigt werden das viele der OWASP Top 10 Schwachstellen mit einem Tool wie Postman überprüft werden können.Die Authentifizierung am Gateway ist momentan auf den OIDC Flow „Password“ eingeschränkt und sollte in Zukunft für die Verwendung von Single Page Applikation erweitert werden auf den OIDC Flow „Authorization Code Flow + PCKE“. Dazu muss das erstellte Javascript modifiziert und getestet werden.

Die Schwachstellen Tests, welche mit Prio 2 bewertet worden sind, konnten im Rahmen der Arbeit nicht mehr umgesetzt werden und könnten ebenfalls in Zukunft noch erstellt werden.

Das Projekt wird eventuell nach der Bachelorarbeit weitergeführt, um noch andere API-Testing Tools wie beispielweise Appknox oder ReadyAPI an der entwickelten API auszutesten.

5 Glossar

Abkürzung	Erklärung
API	Application Programming Interface. Für weitere Informationen siehe [61]
JWT	JSON Web Token (JWT) Zugriffstoken entsprechen dem JWT-Standard und enthalten Informationen über einen User in Form von Claims.
Gitlab	GitLab ist eine webbasierte Versionsverwaltung Applikation für Softwareprojekte. Gitlab bietet für die kontinuierliche Integration die sogenannte GitLab CI an.
JSON	JavaScript Object Notation, Datenformat für einfachen Datenaustausch
Ci/CD	CI/CD ist ein Akronym in der Softwareentwicklung für die Kombination von Continuous Integration (CI) und Continuous Delivery (CD).. Es ist ein sehr nützlicher agiler Prozess für Teams. Eine effektive CI/CD-Pipeline erleichtert Fehlerbehebungen, verrringert Merge-Konflikte und beschleunigt den Entwicklungsprozess.
UI	Die Benutzeroberfläche oder auch Benutzerschnittstelle ist die grafische Oberfläche, mit welcher die Applikation bedient werden kann.
FURPS	FURPS ist ein Akronym aus der Softwarequalität. Die einzelnen Buchstaben stehen jeweils für ein Qualitätsattribut, z.B. F für Functionality.
URI	Uniform Resource Identifier, Identifikator einer eindeutigen Ressource
Endpoint	Ein API-Endpunkt ist ein Punkt, an dem eine API eine Verbindung mit einem anderen Teilnehmer herstellt. APIs funktionieren, indem sie Anforderungen für Informationen von einer Webanwendung oder einem Webserver senden und eine Antwort erhalten.
OAuth2	OAuth2.0 ist ein Autorisierungsprotokoll, das den Zugriff auf Ressourcen regelt. Es funktioniert, indem es die Benutzerauthentifizierung an den Service delegiert, der das Benutzerkonto hostet, und andere Anwendungen (Clients) für den Zugriff auf das Benutzerkonto autorisiert. OAuth2 bietet Autorisierungsabläufe für Web- und Desktop-Anwendungen sowie für mobile Geräte.
Authentifizierung	Authentifizierung ist ein Prozess zur Identifizierung einer Person, in der Regel basierend auf einem Benutzernamen und einem Passwort. Es geht darum zu wissen, dass der Benutzer der Besitzer des Kontos ist.
Autorisierung	Autorisierung ist der Prozess, jemandem die Erlaubnis zu geben, etwas zu tun. Es benötigt die gültige Identifikation des Benutzers, um zu prüfen, ob dieser Benutzer autorisiert ist oder nicht.
OIDC	OpenID Connect (OIDC) ist eine dünne Protokollschicht, die auf OAuth 2.0 aufsetzt und Anmelde- und Profilinformationen über die eingeloggte Person hinzufügt. Das Einrichten einer Anmeldesitzung wird als Authentifizierung bezeichnet, und die Informationen über die angemeldete Person (d. h. den Ressourcenbesitzer) werden als Identität bezeichnet.
Identity Provider	Wenn ein Authentifizierung Service OIDC unterstützt, wird er oft als Identity Provider bezeichnet, da er Informationen über den Besitzer an den Client zurückgibt.

OWASP	Das Open Web Application Security Project (OWASP) ist eine gemeinnützige Stiftung, die sich für die Verbesserung der Sicherheit von Software einsetzt. Durch von der Community geführte Open-Source-Softwareprojekte und zehntausenden von Mitgliedern und führende Bildungs- und Schulungskonferenzen ist die OWASP Foundation die Quelle für Entwickler und Technologen, um das Web zu sichern.
Opaque-Token	Opaque bedeutet übersetzt undurchsichtig. Opaque Zugriffstoken sind Tokens in einem Format, auf das man nicht zugreifen kann, und enthalten in der Regel einen Bezeichner für Informationen im dauerhaften Speicher eines Servers.
ORM-Mapper	Das objektrelationale Mapping (ORM, O/RM und O/R-Mapping-Tool) ist in der Informatik eine Programmiertechnik zur Konvertierung von Daten zwischen inkompatiblen Typsystemen mit objektorientierten Programmiersprachen. Dadurch wird quasi eine "virtuelle Objektdatenbank" geschaffen, die aus der Programmiersprache heraus verwendet werden kann.
REST	REST ist die Abkürzung für Representational State Transfer. Es ist ein architektonischer Stil, welcher eine Reihe von Regeln definiert, wie sich die Architektur eines verteilten -Systems im Internet verhalten sollte.
SQL	Die Abkürzung SQL steht für Structured Query Language und bezeichnet eine Sprache für die Kommunikation mit relationalen Datenbanken.

6 Literaturverzeichnis

[1] S. Bankiervereinung, „Swissbanking," 2020. [Online]. Available: https://www.swissbanking.org/de/themen/digitalisierung/openbanking.

[2] D. Z. ,. J. D. Mark O'Neill, „Gartner - API Security," [Online]. Available: https://www.gartner.com/en/documents/3956746/api-security-what-you-need-to-do-to-protect-your-apis.

[3] M. Korolov, „What you need to know about the new OWASP API Security Top 10 list," [Online]. Available: https://www.csoonline.com/article/3452747/what-you-need-to-know-about-the-new-owasp-api-security-top-10-list.html.

[4] Akamai, „Akamai State Of The Internet Security Report: Retailers Most Common Credential Stuffing Attack Victim; Points To Dramatic Rise In API Traffic As Key Trend," [Online]. Available: https://www.akamai.com/us/en/about/news/press/2019-press/state-of-the-internet-security-retail-attacks-and-api-traffic.jsp.

[5] „Stackify What is Spring Boot?," 2019. [Online]. Available: https://stackify.com/what-is-spring-boot/.

[6] D. M. Kuhrmann, „Rational Unified Process (RUP)," [Online]. Available: https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Vorgehensmodell/Rational-Unified-Process-%28RUP%29.

[7] A. Aldaine, „Top 10 API Testing Tools in 2020," [Online]. Available: https://medium.com/@alicealdaine/top-10-api-testing-tools-rest-soap-services-5395cb03cfa9.

[8] Katalon, „A Review of Top 3 API Tools," [Online]. Available: https://www.katalon.com/resources-center/blog/soapui-vs-postman-katalon-api-tools/.

[9] P. Dutta, „How to Create Your Own API Automation Framework Using Rest Assured," [Online]. Available: https://dev.to/promode/api-automation-framework-using-rest-assured-49jd.

[10] J. Colantonio, „How to Choose an API Testing Tool," [Online]. Available: https://testguild.com/how-to-choose-api-testing-tool/.

[11] Katalon, „Problems with Authentification," [Online]. Available: https://forum.katalon.com/t/problems-with-authentication-tab-of-web-service-request-oauth2/31692.

[12] A. H. Karam, „Continuous integration with Gitlab," [Online]. Available: https://docs.katalon.com/katalon-studio/docs/continuous_integration_gitlab.html.

[13] M. Fowler, „Microservices," [Online]. Available: https://martinfowler.com/articles/microservices.html.

[14] E. Boersma, „Top 10 security traps to avoid when migrating from a monolith to microservices," [Online]. Available: https://blog.sqreen.com/top-10-security-traps-to-avoid-when-migrating-from-a-monolith-to-microservices/.

[15] S. Kappagantula, „Dzone Microservice Architecture," [Online]. Available: https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a.

[16] „Owasp API Security Project," 2019. [Online]. Available: https://owasp.org/www-pdf-archive/API_Security_Top_10_RC_-_Global_AppSec_AMS.pdf.

[17] Wikipedia, „XML-RPC," [Online]. Available: https://en.wikipedia.org/wiki/XML-RPC.

[18] „Soap vs Rest," [Online]. Available: https://raygun.com/blog/soap-vs-rest-vs-json/.

[19] Redhat, „What is a REST API?," [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[20] I. Novikov, „What's Under The Hood Of API Security?," [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2020/07/21/whats-under-the-hood-of-api-security/?sh=13e6f4bd54d9.

[21] M. Khorev, „How to detect and manage web bots?," [Online]. Available: https://wire19.com/how-to-detect-and-manage-web-bots/.

[22] M. McKeay, „State of the Internet Sicherheitsbericht 2020: Finanzdienstleistungen," [Online]. Available: https://www.akamai.com/de/de/multimedia/documents/state-of-the-internet/soti-security-financial-services-hostile-takeover-attempts-report-2020.pdf.

[23] S. Security, „Salt Security Report Reveals API Security Concerns are Inhibiting Business Innovation," [Online]. Available: https://salt.security/press-releases/salt-security-report-reveals-api-security-concerns-are-inhibiting-business-innovation.

[24] Fortinet, „DoS vs DDoS," [Online]. Available: https://www.fortinet.com/resources/cyberglossary/dos-vs-ddos.

[25] S. Cook, „DDoS attack statistics and facts for 2018-2021," [Online]. Available: https://www.comparitech.com/blog/information-security/ddos-statistics-facts/.

[26] Katalon, „Introduction to API Testing," [Online]. Available: https://docs.katalon.com/katalon-studio/docs/introduction_api_testing.html#where-is-api-testing-performed.

[27] M. S. U. Z. D. L. C. P. Olaf Zimmermann, „Microservice API Patterns Responsiblity," [Online]. Available: https://microservice-api-patterns.org/patterns/responsibility/.

[28] Venify, „What Are SSL Stripping Attacks?," [Online]. Available: https://www.venafi.com/blog/what-are-ssl-stripping-attacks.

[29] Keycloak, „Keycloak," [Online]. Available: https://www.keycloak.org/.

[30] Tyk, „Tyk," [Online]. Available: https://tyk.io/.

[31] OWASP, „Zed Attack Proxy," [Online]. Available: https://www.zaproxy.org/.

[32] Snyk, „Snyk," [Online]. Available: https://snyk.io/.

[33] Junit, „Junit," [Online]. Available: https://junit.org/junit4/.

[34] eclemma, „Jacoco," [Online]. Available: https://www.eclemma.org/jacoco/.

[35] M. Wadhwa, „A Beginners Guide To The STRIDE Security Threat Model," [Online]. Available: https://www.ockam.io/learn/blog/introduction_to_STRIDE_security_model.

[36] Owasp, „Threat Dragon," [Online]. Available: https://owasp.org/www-project-threat-dragon/.

[37] NIST, „CVSS Vulnerability Metrics," [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss.

[38] IETF, „RFC 6819 OAuth 2.0 Threat Model and Security Considerations," [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6819.

[39] I. S. Erez Yalon, „OWASP API Security Project," [Online]. Available: https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf.

[40] Cloudfare, „Was ist Credential Stuffing?," [Online]. Available: https://www.cloudflare.com/de-de/learning/bots/what-is-credential-stuffing/.

[41] D. Balaban, „Testing OWASP's Top 10 API Security Vulnerabilities," [Online]. Available: https://nordicapis.com/testing-owasps-top-10-api-security-vulnerabilities/.

[42] Tyk.io, „Tyk and OWASP Top Ten Threats," [Online]. Available: https://tyk.io/docs/basic-config-and-security/security/owasp-top-ten/.

[43] J. Leyden, „Zoom fixes flaws that allowed brute-force attacks to crack private meeting passwords," [Online]. Available: https://portswigger.net/daily-swig/zoom-fixes-flaws-that-allowed-brute-force-attacks-to-crack-private-meeting-passwords.

[44] Wikipedia, „FURPS," [Online]. Available: https://de.wikipedia.org/wiki/FURPS.

[45] Wikipedia, „SMART," [Online]. Available: https://de.wikipedia.org/wiki/SMART_(Projektmanagement).

[46] Spring, „RestDocs," [Online]. Available: https://spring.io/projects/spring-restdocs.

[47] https://springframework.guru/, „Bean Validation in Spring Boot," [Online]. Available: https://springframework.guru/bean-validation-in-spring-boot/.

[48] „Mapstruct.org," [Online]. Available: https://mapstruct.org/.

[49] „Content Security Policy (CSP)," [Online]. Available: https://content-security-policy.com/unsafe-inline/.

[50] IETF, „RFC 7519 JWT," [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519.

[51] jwt.io, „JWT Introduction," [Online]. Available: https://jwt.io/introduction.

[52] Curity.io, „The Split Token Approach," [Online]. Available: https://curity.io/resources/learn/split-token-pattern/.

[53] „Validate Access Tokens," [Online]. Available: https://auth0.com/docs/tokens/access-tokens/validate-access-tokens.

[54] IETF, „OAuth 2.0 Scope," [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6749#section-3.3.

[55] Jacoco, „JaCoCo Java Code Coverage Library," [Online]. Available: https://www.eclemma.org/jacoco/.

[56] D. Roccasalva, „Stealing JWTs in localStorage via XSS," [Online]. Available: https://medium.com/redteam/stealing-jwts-in-localstorage-via-xss-6048d91378a0.

[57] „jsonschema.net," [Online]. Available: https://jsonschema.net/home.

[58] D. Berman, „Secure coding with Snyk's new JetBrains IDE plugin," [Online]. Available: https://snyk.io/blog/secure-coding-with-jetbrains-ide-plugin/.

[59] „Should I use CSRF protection on Rest API endpoints?," [Online]. Available: https://security.stackexchange.com/questions/166724/should-i-use-csrf-protection-on-rest-api-endpoints/166798#166798.

[60] OWASP, „ZapProxy Getting Stared," [Online]. Available: https://www.zaproxy.org/getting-started/.

[61] „How can you use ZAP to scan APIs?," [Online]. Available: https://www.zaproxy.org/faq/how-can-you-use-zap-to-scan-apis/.

[62] Redhat, „Was ist eine API?," [Online]. Available: https://www.redhat.com/de/topics/api/what-are-application-programming-interfaces.

[63] HSR, „GitLab Hsr," [Online]. Available: https://gitlab.dev.ifs.hsr.ch/.

[64] [Online].

[65] Inflectra, „What is API Testing?," [Online]. Available: https://www.inflectra.com/rapise/highlights/api-testing.aspx.

[66] I. Wigmore, „Techtarget Monolithic Architecture," [Online]. Available: https://whatis.techtarget.com/definition/monolithic-architecture.

[67] „Techopedia Microservices,“ [Online]. Available: https://www.techopedia.com/definition/32503/microservices.

[68] B. Brodie, „DZone Increase Security,“ [Online]. Available: https://dzone.com/articles/increase-security-by-transitioning-from-monolith-t.

[69] „Divante Monolithic vs Microservices,“ [Online]. Available: https://divante.com/blog/monolithic-architecture-vs-microservices/.

[70] I. Shkedy, „Medium: Modern Application Security — What are Modern Applications?,“ [Online]. Available: https://inonst.medium.com/modern-application-security-what-are-modern-applications-597fdac082a.

[71] I. Sacolick, „What is CI/CD,“ [Online]. Available: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html.

[72] Versprite, „Application Threat Modeling,“ [Online]. Available: https://versprite.com/security-offerings/appsec/application-threat-modeling/.

[73] „Common Vulnerability Scoring System Version 3.1,“ [Online]. Available: https://www.first.org/cvss/v3-1/.

7 Abbildungen

8 Erklärung zur Urheberschaft

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne Hilfe Dritter angefertigt habe. Ich habe nur die Hilfsmittel benutzt, die ich angegeben habe. Gedanken, die ich aus fremden Quellen direkt oder indirekt übernommen habe, sind kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort/Datum

Rapperswil, 18. Juni 2021

Unterschrift

Maximilian Marxer

Anhang

I Benutzerhandbuch

Installationsanleitung:

Die Installationsanleitung für Android Smartphones ist als separate Anlage verfügbar.

II Projektplan

Der Projektplan ist als PDF-Datei als separate Anlage verfügbar.

III Postman Test Report

Der Postman Test Report ist als HTML-Datei als separate Anlage verfügbar.

IV Postman Test Collection

Die Postman Test-Collections sind als JSON Dateien separat verfügbar

V Rest Docs Dokumentation

Die erstellte Dokumentation ist als HTML Datei als separate Anlage verfügbar

VI Jacoco Test Coverage Report

Der Test Coverage Report ist als HTML Datei als separate Anlage verfügbar

VII Script für Opaque Token

Das erstellte Javascript für die Erstellung des Opaque Tokens im Gateway ist als separate Anlage verfügbar

VIII Zeitauswertung

Zeitauswertung nach Sprints:

Nachfolgende Grafiken zeigen die Auswertungen der geplanten und benötigten Zeiten während der einzelnen Sprints in diesem Projekt.

	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	Sprint 7	Sprint 8
Zeit geplant (h)	4.00	51.00	51.00	50.00	51.00	51.00	51.00	51.00
Zeit effektiv (h)	4.00	52.50	55.00	45.00	49.00	55.00	53.00	53.00

Zeit geplant (h)	360.00
Zeit effektig (h)	366.50

Abbildung 81 Zeitauswertung Detail

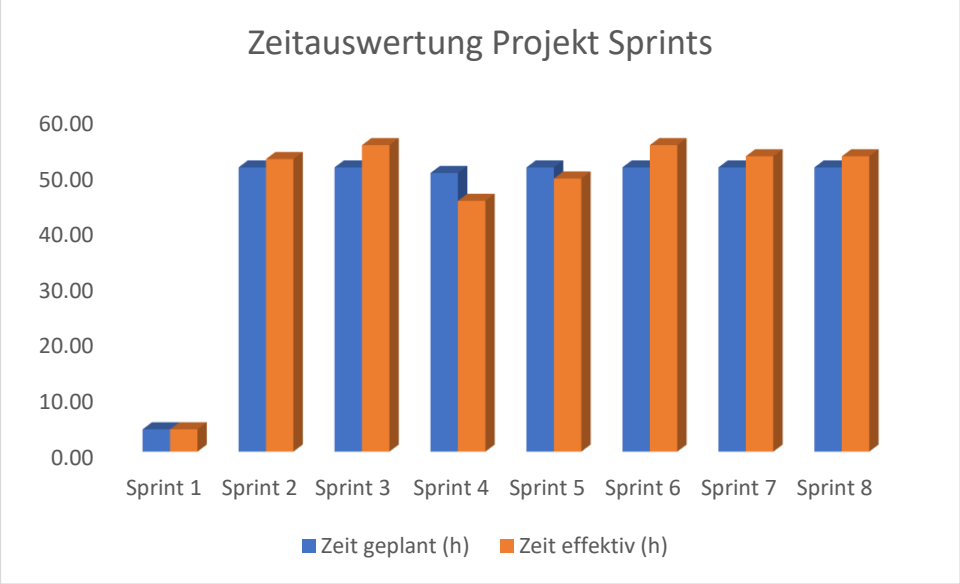


Abbildung 82 Zeitauswertung Projekt-Sprints