HAW HAMBURG

BACHELOR THESIS
Ann-Kathrin Bales

XSS in Rust-Webanwendungen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Ann-Kathrin Bales

XSS in Rust-Webanwendungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Klaus-Peter Kossakowski

Eingereicht am: 2024-09-27

Ann-Kathrin Bales

Thema der Arbeit

XSS in Rust-Webanwendungen

Stichworte

XSS, Cross-Site Scripting, Rust, Webanwendungen, Sicherheit, OWASP Juice Shop

Kurzzusammenfassung

Seit Aufkommen des Web 2.0 stellen Angriffe durch Cross-Site Scripting (XSS) ein verbreitetes Problem dar und werden als eine der Hauptbedrohungen für Webanwendungen eingestuft. Die relativ neue, sicherheitsorientierte Programmiersprache Rust wird zunehmend auch in Webanwendungen eingesetzt und geht damit über ihren ursprünglichen Einsatzbereich in der systemnahen Programmierung auf Browser- oder Betriebssystemebene hinaus. In dieser Arbeit werden die Wirkungsweise von XSS-Angriffen auf Rust-Webanwendungen und mögliche Vorteile der Sprache im Schutz gegen diese betrachtet. Ein Prototyp einer in Rust entwickelten Webanwendung nach dem Vorbild des *OWASP Juice Shop* wird durch XSS angegriffen und dadurch gezeigt, dass Rust allein, ohne den Einsatz weiterer Maßnahmen, nicht gegen Cross-Site Scripting schützen kann.

Ann-Kathrin Bales

Title of Thesis

XSS in Rust Web Applications

Keywords

XSS, Cross-Site Scripting, Rust, web applications, security, OWASP Juice Shop

Abstract

Cross-Site Scripting (XSS) attacks have been a prevalent problem since the rise of the web 2.0 and are consistently ranked among the main threats against web applications. More recently, the relatively new, safety-focused programming language Rust is gaining

popularity for use in building web applications, expanding from its original purpose as a systems programming language in the context of operating systems and browsers. This thesis investigates the mechanisms of XSS attacks on web applications in Rust and possible benefits of the language in protecting against them. A prototype of a Rust web application modeled after *OWASP Juice Shop* is attacked using XSS, showing that Rust cannot protect against Cross-Site Scripting without employing additional tools.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

API Application Programming Interface.

CI Continuous Integration.

CSP Content Security Policy.

CWE Common Weakness Enumeration.

DOM Document Object Model.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

IDE Integrated Development Environment.

MIR Mid-level Intermediate Representation.

ORM Object Relational Mapper.

OWASP Open Worldwide Application Security Project.

TLS Transport Layer Security.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

XSS Cross-Site Scripting.

1 Einleitung

Seit Beginn seiner Entwicklung im Jahr 2006 ist Rust als „sichere Sprache“ im Gespräch und gehört seit der Veröffentlichung von Version 1.0 im Jahr 2015 zu den bei Entwickler:innen beliebtesten Sprachen (vgl. Abschnitt 4.2). Zunehmend rückt dabei auch die Verwendung in der Webentwicklung in den Fokus. Durch die Digitalisierung aller Gesellschaftsbereiche nehmen Webanwendungen wichtige Funktionen wahr und so ist für ihre Entwicklung eine Sprache wie Rust interessant, die einen Fokus auf Sicherheit und Nebenläufigkeit setzt. Es stellt sich die Frage: Bieten die Vorzüge von Rust auch Schutz vor typischen Schwachstellen von Webanwendungen?

Im Folgenden soll dieser Frage am Beispiel der verbreiteten Angriffsart Cross-Site Scripting nachgegangen werden. Um geeignete Angriffe auszuwählen, wird die angreifbare Webanwendung *OWASP Juice Shop* analysiert und deren XSS-Teil in einem in Rust geschriebenen Prototyp umgesetzt. Dabei wird deutlich, dass Rust keine eigenen Abwehrmechanismen gegen diese Art von Angriffen besitzt. Somit ist bei der Auswahl von Herangehensweisen und zusätzlichen Sicherheitswerkzeugen die gleiche Sorgfalt wie bei der Verwendung anderer Sprachen notwendig, wann immer Nutzer:inneneingaben in einer Anwendung verwendet werden.

Im ersten Schritt wird auf die Sicherheit von Webanwendungen allgemein eingegangen und Angriffe auf diese werden im Kontext der von ihnen verarbeiteten und verwalteten Güter betrachtet. Anschließend wird die Funktionsweise von Cross-Site Scripting analysiert und die Hauptarten von XSS werden beschrieben. Rust wird als Sprache vorgestellt, wobei besonders die Idee einer „sicheren Sprache“ sowie die Verwendung von Rust thematisiert werden. Einige verbreitete Frameworks für den Einsatz von Rust in Webanwendungen sowie eine Auswahl allgemeiner und XSS-spezifischer Sicherheitstools werden ebenfalls vorgestellt. Die zu Trainings- und Demonstrationszwecken angreifbare Webanwendung *OWASP Juice Shop* wird in ihrem Aufbau und den Teilen, die XSS-Angriffe abdecken, analysiert. Im Anschluss wird ein Prototyp einer angreifbaren Anwendung in

Rust unter Verwendung eines der zuvor vorgestellten Web-Frameworks geplant und entwickelt. Diese orientiert sich am Juice Shop, beschränkt sich aber auf den Bereich XSS. Der Erfolg der dort umgesetzten Angriffe wird bewertet und ein Vergleich mit dem Juice Shop gezogen.

2 Sicherheit von Webanwendungen

Um Webanwendungen in ihrem Kontext beschreiben und verstehen zu können, müssen zunächst die Begriffe *Internet* und *World Wide Web* (kurz häufig *Web*) definiert werden. Als *Internet* wird das weltweite Netzwerk verbundener Computer und verwandter Geräte bezeichnet (auch wenn diese Beschreibung eher den Begriff World Wide Web nahelegt) – [Pfleeger u. a. 2023] vergleichen es mit einem Bibliotheksgebäude. Innerhalb dieses Bibliotheksgebäudes entspricht das World Wide Web der Menge von Büchern: Es ist eine Sammlung von Webseiten, die auf Servern gespeichert sind und von Clients wie etwa Webbrowsern abgerufen werden können. Eine Webseite ist nach [Hoffman 2024] definiert als über das Internet (meist per HTTP) zugängliche Menge von Dokumenten mit Informationen. Eine Webanwendung ist das Gegenstück zu einer nativen Anwendung, das aber in einem Browser statt direkt auf einem Betriebssystem läuft – mit Nutzer:innenrollen, Interaktionsmöglichkeiten und der Möglichkeit zum Speichern sowie Abrufen von Daten für Benutzer:innen [Hoffman 2024]. Das Internet und das darüber bereitgestellte Web sind von enormer weltweiter Bedeutung: 2022 hatten von ca. acht Milliarden Menschen auf der Welt rund fünf Milliarden Zugang zum Internet und konnten fast zwei Milliarden Webseiten aufrufen [Pfleeger u. a. 2023]. Entsprechend wichtig ist auch die Sicherheit von Webanwendungen, die schützenswerte Güter wie sensible Daten zahlreicher Menschen, Organisationen und Firmen verwalten.

2.1 Schutzziele

Die klassischen Schutzziele in der IT-Sicherheit sind *Informationsvertraulichkeit* (*confidentiality*), *Datenintegrität* (*integrity*) und *Verfügbarkeit* (*availability*) – abgekürzt als CIA [Eckert 2023]. Dabei versteht man unter Informationsvertraulichkeit, dass keine unautorisierte Person Informationen erlangen kann [Eckert 2023]. Datenintegrität bedeutet, dass eine unbefugte und unbemerkte Veränderung von Daten nicht möglich sein darf, und Verfügbarkeit, dass der berechtigte Zugriff auf Daten nicht eingeschränkt ist

[Eckert 2023]. [Eckert 2023] erweitert die Schutzziele um *Authentizität*, *Verbindlichkeit* und *Schutz personenbeziehbarer Daten*. Die Parker-Hexade (*Parkerian hexad*) nach Donn Parker erweitert die Schutzziele ebenfalls um *Authentizität* (*authenticity*); zusätzlich sind hier *Nutzen* (*utility*) und *Besitz* (*possession*) ergänzt [Andress 2014]. *Utility* beschreibt die Nützlichkeit der Daten für eine Person oder Institution – auch für eine:n Angreifer:in [Andress 2014]. *Possession* oder auch *control* bezieht sich auf den Besitz von Daten, etwa auf einem Speichermedium [Andress 2014].

Für Webanwendungen spielen vor allem die Informationsvertraulichkeit und der damit eng verbundene Schutz personenbeziehbarer Daten eine wichtige Rolle, aber auch die weiteren Schutzziele sind relevant. Um ihrer Funktion nachzukommen, verarbeiten die meisten Webanwendungen personenbeziehbare Daten (z. B. Adressen), die entsprechend geschützt werden müssen. Einige Bereiche, in denen Webanwendungen eingesetzt werden, machen hierbei die Verwaltung und Speicherung von besonders schützenswerten Informationen notwendig, wie etwa personenbeziehbaren Gesundheitsdaten. Die Informationsvertraulichkeit muss auch dadurch geschützt werden, dass die Sichtbarkeit von Daten auf autorisierte Personen beschränkt ist – etwa durch Nutzer:innenrollen und Passwörter. Die von Webanwendungen verarbeiteten und bereitgestellten Daten müssen vor unbefugter Veränderung geschützt (*integrity*) und ihre Bereitstellung gesichert werden (*availability*). Für Transaktionen, etwa in Webanwendungen für Onlineshops, ist *authenticity* wichtig, also die korrekte Zuordnung von z. B. Zahlungsdaten zur richtigen Person, und damit auch das angrenzende Konzept von *nonrepudiation*: die Unmöglichkeit, eine Aktion abzustreiten [Andress 2014]. Diese Schutzziele bilden die Grundlage, um Entscheidungen darüber zu treffen, wie Webanwendungen gegen Angriffe abzusichern sind.

2.2 Angriffe auf IT-Systeme

Angriffe auf IT-Systeme, häufig als Hacken (*hacking*) bezeichnet, existieren seit mindestens der Mitte des 20. Jahrhunderts, die Anzahl von Angreifer:innen hat mit der Verbreitung des Internets jedoch stark zugenommen [Hoffman 2024]. Als Angriff bezeichnet man einen Versuch, sich unbefugt Zugang zu einem System zu verschaffen, um auf dessen Ressourcen zuzugreifen oder es zu stören [Stouffer u. a. 2023].

In den Anfängen des World Wide Web in den 1990er Jahren bestand dessen Zweck in erster Linie darin, statische HTML-Seiten bereitzustellen, die Nutzer:innen nur konsumierten. Dies änderte sich in den frühen 2000ern, als Internetseiten interaktiver wurden

und Nutzer:innen erlaubten, Inhalte hinzuzufügen oder zu verändern – es wurde zum sogenannten *Web 2.0* [Hoffman 2024]. JavaScript etablierte sich als de-facto Sprache für Webbrowser und mit Skriptsprachen konnte HTML dynamisch generiert werden [McDonald 2020]. Diese Veränderung in der Funktionsweise des Web verwandelte es laut [Hoffman 2024, S. 13] „from a document-sharing platform to an application distribution platform“ und eröffnete Hacker:innen eine neue Kategorie von Angriffen, darunter auch Cross-Site Scripting (XSS). Statt nur Netzwerke und Server anzugreifen, konnte nun auch auf Nutzer:innen gezielt werden. [McDonald 2020, S. XXI] resümiert hierzu: „the internet became a much more dangerous place“. Firmen und andere Organisationen versuchten vor dem Aufkommen von Webanwendungen größtenteils, ihre Sicherheit am Netzwerkrand zu sichern. Dies hat sich mit dem Einsatz von Webanwendungen in praktisch allen Bereichen des wirtschaftlichen Handelns und des öffentlichen Lebens geändert. So stellen auch [Stuttard und Pinto 2011, S. 12] fest: „the security perimeter of a typical organization has moved. Part of that perimeter is still embodied in firewalls and bastion hosts. But a significant part is now occupied by the organization's web applications“.

2.3 Webanwendungen

Webanwendungen wurden aufgrund der genannten Entwicklungen zu einem attraktiven Angriffsziel und stehen heute im Fokus von Hacker:innen [Hoffman 2024, S. 16]. [Stuttard und Pinto 2011] begründen die Verbreitung von Angriffen auf die Nutzer:innen von Webanwendungen damit, dass es lukrativ und einfach sei – es lohne sich, mit einem simplen Angriff ein Prozent der Nutzer:innen einer Bank zu erreichen, statt die Bank selbst aufwändig anzugreifen.

Für die Einordnung von Sicherheitsproblemen für Webanwendungen ist die als *OWASP Top Ten* veröffentlichte Liste der wichtigsten Schwachstellen von und Angriffen auf Webanwendungen des *Open Web Application Security Project* (OWASP) zentral [Eckert 2023]. Das gemeinnützige Projekt wurde von Firmen und Organisationen (u. a. IBM, British Telecom, Swiss Federal Institute of Technology) aus dem Bereich der Informationstechnologie gegründet, um Webanwendungen sicherer zu machen [Eckert 2023]. In dieser Liste ist auch XSS vertreten, dies wird in Abschnitt 3.1 weiter ausgeführt.

Auch die Liste der *CWE Top 25 Most Dangerous Software Weaknesses*, die von der von der *U.S. Department of Homeland Security Cybersecurity and Infrastructure Security Agency* finanziert und durch die MITRE Corporation erstellt wird, ist ein wichtiges

Werkzeug für die Bewertung von Schwachstellen [The MITRE Corporation 2023]. Hier ist XSS ebenfalls vertreten, seit 2010 unter dem Namen *Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')* [The MITRE Corporation c].

Für den Zugriff auf Webanwendungen ist der Webbrowser (oder kurz Browser) zentral – so zentral, dass laut [Alcorn u. a. 2014] zahlreiche Nutzer:innen ohne technischen Hintergrund ihn als „das Internet" ansehen. Dabei kann der Browser unter anderem ohne Einflussnahme der Person, die ihn verwendet, Anweisungen von der anzuzeigenden Webseite – und anderen Seiten, etwa für die Einbindung von externen Inhalten wie Anzeigen – anfordern und ausführen. Dies ist für seine Funktion notwendig, eröffnet aber auch Angriffsmöglichkeiten. Interessant werden solche Angriffe für Hacker:innen gerade durch die Verwendung von Browsern für alle denkbaren Zwecke, von Internetbanking über Social-Media-Seiten bis hin zu firmeninternen Anwendungen [Alcorn u. a. 2014]. Hier liegt auch ein Problem für die Sicherheit von Webanwendungen: Selbst wenn der Server gut abgesichert ist, können auf Seiten des Clients verschiedene Technologien in verschiedenen, auch veralteten Versionen mit gegebenenfalls bekannten Schwachstellen eingesetzt werden, was zahlreiche Angriffsvektoren eröffnet [Stuttard und Pinto 2011].

Um die Struktur einer Webseite zu repräsentieren, verwendet der Webbrowser eine Datenstruktur namens *Document Object Model (DOM)* [McDonald 2020]. Diese umfasst eine API, die das HTML in anzuzeigenden Webseiten organisiert, sowie APIs für die Verwaltung von URLs, Cookies, der Browserhistorie und weiteren Browserfunktionen [Hoffman 2024]. Sie wird im Arbeitsspeicher gehalten und kann durch JavaScript direkt manipuliert werden, sodass Veränderungen in einer angezeigten Seite ohne Interaktion mit einem Server möglich sind [McDonald 2020; Alcorn u. a. 2014]. Dadurch werden DOM-basierte XSS-Angriffe möglich, die in Unterabschnitt 3.2.3 weiter behandelt werden [McDonald 2020].

Für die Erfüllung ihrer Funktion speichern Webanwendungen Informationen, die schützenswerte Güter (*assets*) darstellen. Dazu gehören etwa Logininformationen, Benutzer:innendaten oder Informationen zu Zahlungsmitteln, die lohnende Ziele für Angriffe sind. Hierbei haben (gerade kostenlose) Webanwendungen eine besondere Stellung im Vergleich mit anderer Software, da sie unmittelbar für Millionen von Nutzer:innen zugänglich sind – also auch für Hacker:innen [McDonald 2020]. Webanwendungen verarbeiten auf viele verschiedene Arten und in verschiedenen Kontexten Nutzer:inneneingaben und kommunizieren mit Back-End-Systemen, die häufig businesskritische oder anderweitig wertvolle Daten enthalten – sie stellen also ein potentielles Einfallstor für Angriffe auf

diese Daten dar [Stuttard und Pinto 2011]. [Stuttard und Pinto 2011] zufolge erfolgen die meisten Angriffe auf Webanwendungen dadurch, dass durch eine Eingabe auf einem Server ein nicht vorhergesehenes oder nicht erwünschtes Verhalten ausgelöst wird.

Gleichzeitig sind Webanwendungen – inklusive der Eingabemöglichkeiten – schwer verzichtbar, sowohl für die Firmen, die für die Bereitstellung ihrer Dienste und schlussendlich für die Erwirtschaftung ihrer Gewinne auf sie angewiesen sind, als auch für die Nutzer:innen, die solche Dienste in Anspruch nehmen wollen und zunehmend weniger (zumindest gleichwertige) Alternativen haben. Für beide Seiten geht es bei der Sicherheit von Webanwendungen um wichtige Güter – und für Angreifer:innen genauso [Stuttard und Pinto 2011].

3 XSS

Cross-Site Scripting (XSS) (auch Cross-site Scripting oder Cross Site Scripting) ist eine Kategorie von Angriffen auf Webanwendungen, bei denen durch ungeprüft verwendete Eingaben Schadcode eingebracht wird [OWASP Foundation b]. [Pfleeger u. a. 2023, S. 288] fassen die Funktionsweise als „passing commands disguised as input“ zusammen. Damit machen sich diese Angriffe das zunutze, was [Stuttard und Pinto 2011, S. XXV] als das Kernproblem von Webanwendungen bezeichnen: „Vulnerabilities in web applications arise because of a single core problem: users can submit arbitrary input.“ Auch [Hoffman 2024] führt die Verbreitung von XSS auf die hohe Anzahl von Interaktionen mit Nutzer:innen zurück und warnt, dass jedes dynamisch erzeugte Script, das ausgeführt wird, eine Webanwendung dem Risiko der Modifikation öffne. XSS macht sich dabei die Tatsache zunutze, dass Browser JavaScript ausführen, wann immer sie es auf einer Webseite finden. Dass JavaScript dabei jeden Teil der Webseite verändern kann, bietet Angreifer:innen zahlreiche Möglichkeiten [McDonald 2020]. Wie in Kapitel 2 aufgezeigt, sind Angriffe auf Webanwendungen für Hacker:innen attraktiv – darunter auch XSS-Angriffe.

Der Begriff Cross-Site-Scripting reflektiert [Pfleeger u. a. 2023, S. 296] zufolge die Interaktion zwischen Gerät und Webseite: „it means that the attack script crosses from a site back to your device“. Anfangs wurde auch die Abkürzung CSS verwendet, die jedoch aufgrund der möglichen Verwechslung mit *Cascading Style Sheets* durch XSS verdrängt wurde [Alcorn u. a. 2014].

Es werden drei grundlegende Arten von XSS unterschieden, die in Abschnitt 3.2 definiert werden: Stored, Reflected und DOM-based XSS.

3.1 Bedeutung

Cross-Site Scripting gehört zu der in Abschnitt 2.2 angesprochenen neuen Kategorie von Angriffen auf Nutzer:innen, die durch durch das Web 2.0 entstanden ist [Hoffman 2024]. Im Februar 2000 meldete das *Computer Emergency Response Team Coordination Center* der amerikanischen Carnegie Mellon University einen der ersten belegten XSS-Angriffe und verwendete bereits die Formulierung „cross-site“ [Alcorn u. a. 2014]. In den 2000ern verbreiteten sich XSS-Angriffe stark [Hoffman 2024]. Einige weit verbreitete Webanwendungen wurden erfolgreich angegriffen, so konnten etwa 2011 durch eine XSS-Schwachstelle im *Android Web Market* (heute *Google App Store*) von Angreifer:innen Anwendungen mit beliebigen Berechtigungen auf dem Gerät eines Opfers installiert werden [Alcorn u. a. 2014]. 2011 schätzten [Stuttard und Pinto 2011] XSS als häufigste Schwachstelle ein, die eine überwältigende Mehrheit von – auch sicherheitskritischen – Webanwendungen betreffe, u. a. Bankanwendungen. Nach wie vor ist XSS unter den OWASP Top Ten der kritischsten Bedrohungen für Webanwendungen vertreten – im Bericht von 2017 als eigenständige Kategorie [OWASP Foundation 2017], im Jahr 2021 gemeinsam mit ähnlichen Angriffen, bei denen ebenfalls Schadcode eingebracht wird, unter *Injection* [OWASP Foundation 2021c]. Dabei stand *XSS* im Jahr 2017 an siebter Stelle, *Injection* auf dem ersten Platz. 2021 wurden Schwachstellen der nun auch XSS umfassenden Kategorie *Injection* in 94 Prozent der untersuchten Softwareanwendungen gefunden und diese nahm damit den dritten Platz hinter *Broken Access Control* und *Cryptographic Failures* ein, wie in 3.1 dargestellt [OWASP Foundation 2021b].

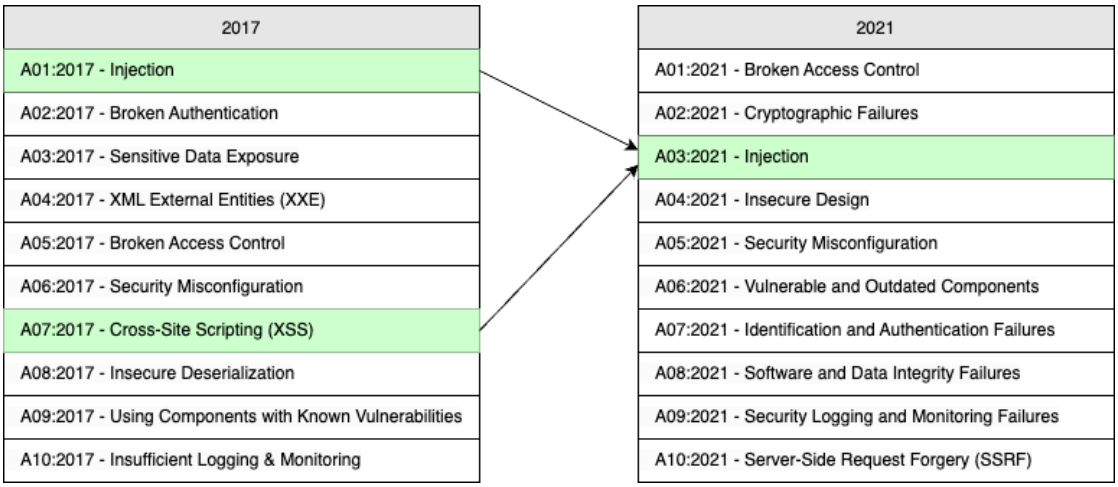

Abbildung 3.1: OWASP Top Ten 2017 und 2021 im Vergleich, nach [OWASP Foundation 2021b], Template: draw.io (CC BY 4.0)

Die 33 Einträge der *Common Weakness Enumeration (CWE)*, die der Kategorie *Injection* zugeordnet wurden, waren 2021 die in den betrachteten Anwendungen insgesamt am zweithäufigsten vertretenen Angriffe [OWASP Foundation 2021b]. 2017 waren die XSS-Angriffe allein die zweithäufigste Form der Angriffe und in zwei Drittel aller getesteten Anwendungen vertreten [OWASP Foundation 2017]. Damit ist XSS für die getesteten Webanwendungen eine der wichtigsten Sicherheitsbedrohungen.

Auch in der von der *U.S. Department of Homeland Security Cybersecurity and Infrastructure Security Agency* finanzierten und durch die MITRE Corporation erstellten Liste der *CWE Top 25 Most Dangerous Software Weaknesses* nahm XSS als *Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')* sowohl 2022 als auch 2023 Platz zwei ein [The MITRE Corporation 2023]. Allgemein wird XSS seit Jahren konsistent als eine der häufigsten Schwachstellen im Web eingestuft [Stuttard und Pinto 2011; Prasad 2016; Najera-Gutierrez 2018; Gebeshuber u. a. 2019; Hoffman 2024]. [Gebeshuber u. a. 2019, S. 272] berichten sogar, es werde häufig als die „größte Sicherheitsbedrohung im Web“ bezeichnet und heben die Gefahr der Kombination mit anderen Schwachstellen hervor, während [Stuttard und Pinto 2011, S. 432] XSS als „the Godfather of attacks against other users“ bezeichnen.

[Alcorn u. a. 2014] ordnen XSS neben Social Engineering und Phishing in ihrer *browser hacking methodology* (vgl. Abbildung 3.2) in der ersten Phase des Vorgehens (*initiating control*) ein. Diese Phase bildet die Grundlage für Angriffe, indem sie Angreifer:innen Zugang zu ihrem Ziel gewährt. Hier liegt der Fokus also auf XSS als Basis für weitere Schritte, etwa unter Ausnutzung einer übernommenen Sitzung (*session*). Auf den ersten Schritt der Kontrollübernahme folgt die Aufrechterhaltung der Kontrolle (*retaining control*), die kontinuierlich geschehen muss, während die eigentlichen Angriffe durchgeführt werden (*attacking*), etwa auf Nutzer:innen, Browser oder Netzwerke.

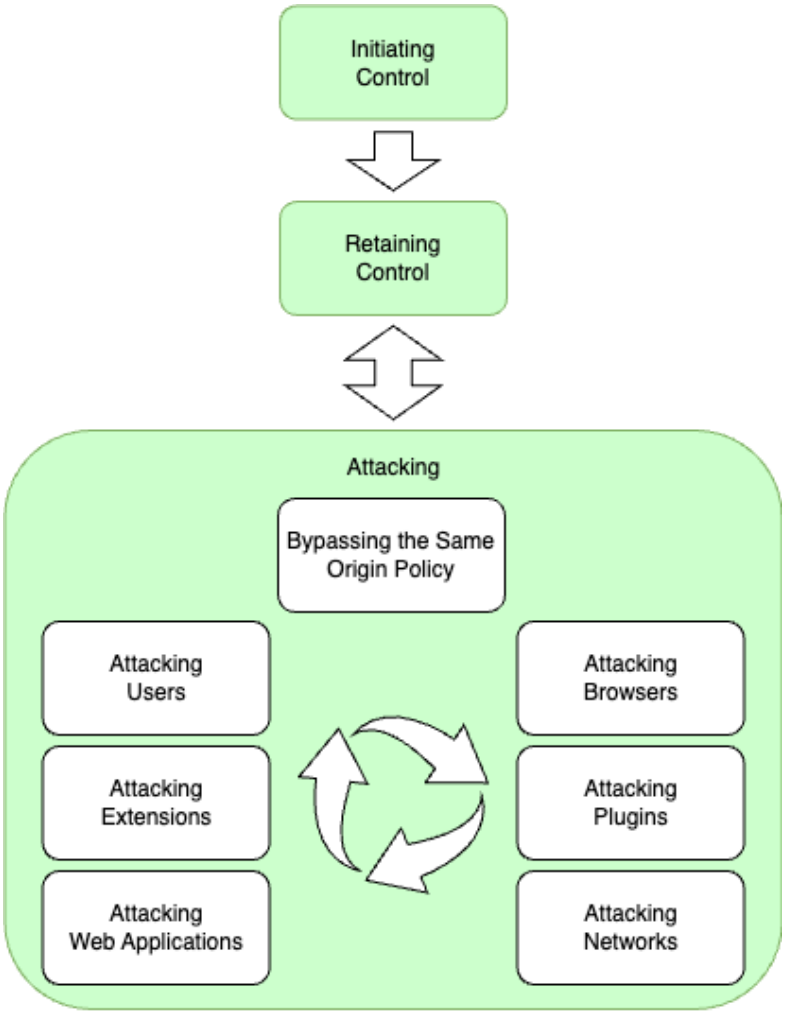

Abbildung 3.2: Methodische Vorgehensweise beim Browser Hacking nach [Alcorn u. a. 2014, S. 23], Template: draw.io (CC BY 4.0)

Alle drei Arten von XSS können von automatisierten, allgemein erhältlichen Werkzeugen ausgeführt werden [OWASP Foundation 2017], was ihren Einsatz wenig zeit- und arbeitsintensiv und auch für ungeschulte Angreifer:innen möglich macht. Dadurch sind nicht nur Webseiten, die von erhöhtem Interesse für Hacker:innen sind, gefährdet, sondern beliebige Webanwendungen. Besonders gefährdet hierfür sind der OWASP Foundation zufolge verbreitet eingesetzte Technologien [OWASP Foundation 2017], da die Entwicklung automatisierter Angriffe hier wahrscheinlich lohnender ist. DOM-based XSS setzt hingegen gute Kenntnisse des Browser-DOM und von JavaScript voraus, was eine größere Hürde

für den Einsatz darstellt, sodass hier nur technisch versierte Angreifer:innen erfolgreich sein können [Hoffman 2024].

[Pfleeger u. a. 2023] kategorisieren XSS-Angriffe als Angriffe, die auf die Gewinnung sensibler Informationen zielen. Durch erfolgreiches Cross-Site Scripting können Angreifer:innen, teilweise ohne dass dies für die angegriffene Seite feststellbar ist, auf alle Daten der Webanwendung zugreifen und Daten an einen anderen Server senden oder von diesem empfangen. Sie können Sitzungen übernehmen und Phishing-Angriffe durchführen, indem sie die grafische Oberfläche manipulieren [Hoffman 2024]. Mögliche Ziele können dabei der direkte Diebstahl sensibler Informationen wie Logindaten oder Kreditkartennummern, das Einfügen eigener Inhalte in die Webseite – etwa zur Verunstaltung (*defacement*) – oder auch die Durchführung weiterer Angriffe sein [McDonald 2020; OWASP Foundation 2017; Stuttard und Pinto 2011; Amberg und Schmid 2024]. [Stuttard und Pinto 2011, S. 433] bezeichnen die Behauptung „You can't own a web application via XSS“ als verbreiteten Mythos und betonen, selbst eine solche Übernahme allein durch XSS in vielen Fällen erfolgreich umgesetzt zu haben.

Die von XSS-Angriffen ausgehende Gefahr schätzt die OWASP Foundation bei reflektierten und DOM-XSS-Attacken als moderat und bei Stored-XSS-Attacken, bei denen Logindaten gestohlen, Sitzungen übernommen oder Malware installiert werden kann, als schwerwiegend ein [OWASP Foundation 2017]. Gefährdet sind vor allem die Informationsvertraulichkeit (auch für personenbeziehbare Daten) und die Datenintegrität (vgl. Abschnitt 2.1), da unbefugte Personen mit Hilfe von XSS auf Daten zugreifen und diese manipulieren können.

3.2 Arten

Basierend auf der Funktionsweise beziehungsweise dem Ort der Verwendung des Schadcodes werden verschiedene Kategorien von XSS-Angriffen unterschieden. Die drei grundlegenden Arten sind *Stored XSS* (Schadcode wird vor der Ausführung in einer Datenbank gespeichert), *Reflected XSS* (Schadcode wird von einem Server reflektiert) und *DOM-based XSS* (Schadcode wird im Browser gespeichert und ausgeführt) [Hoffman 2024; OWASP Foundation e].

3.2.1 Stored XSS

[Hoffman 2024] schätzt Stored-XSS-Angriffe als die häufigste Form ein und als diejenige, die Nutzer:innen am meisten schadet. Da der in eine Datenbank eingeschleuste Schadcode von beliebig vielen Nutzer:innen unwissentlich angefordert werden kann, sind unter Umständen alle Nutzer:innen einer Webanwendung betroffen.

Bei einem Stored-XSS-Angriff werden Eingaben (etwa in einem Feedbackformular) gespeichert, ohne validiert und bereinigt zu werden (Abbildung 3.3, 1.), und werden später an eine andere Person, die möglicherweise sogar administrative Rechte besitzt, gesendet (3.). Der enthaltene Schadcode wird ausgeführt (5.), da der Browser ihn als JavaScript erkennt. Dies kann zum Beispiel dazu führen, dass das Session Token an die angreifende Person gesendet wird (6.), die daraufhin die Session übernehmen kann. [Gebeshuber u. a. 2019; OWASP Foundation 2017; Hoffman 2024; Amberg und Schmid 2024]

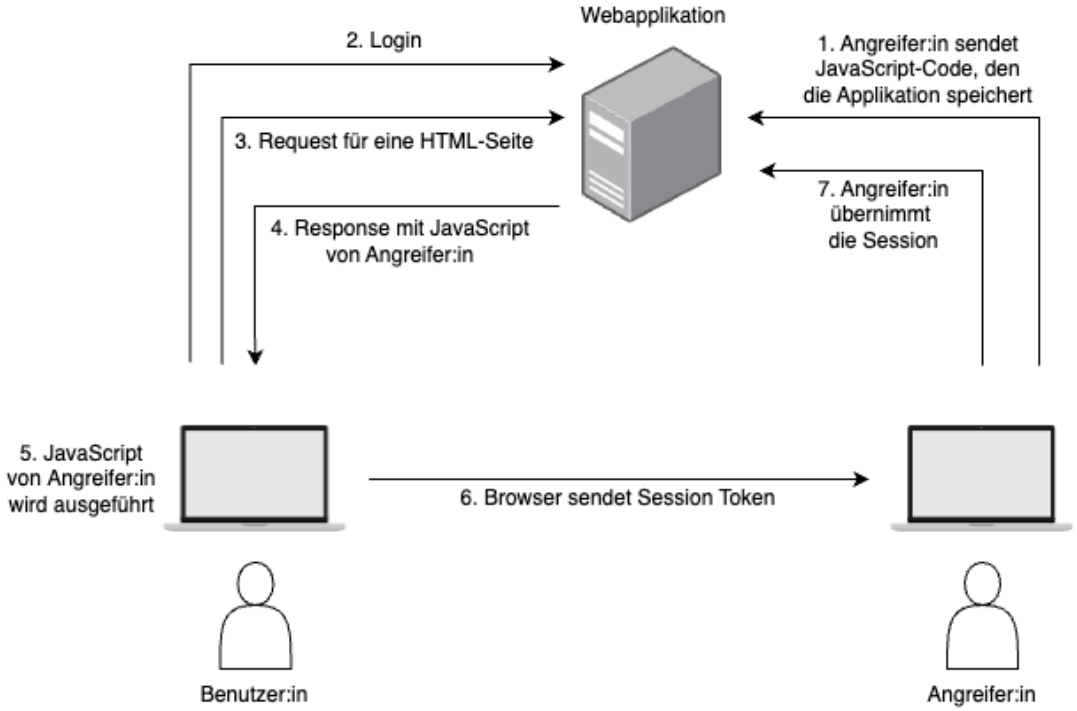

Abbildung 3.3: Stored-XSS-Angriff, nach [Gebeshuber u. a. 2019], Icons: draw.io (CC BY 4.0)

Dieser Typ des Angriffs wird auch als *Type II* oder – aufgrund der Speicherung der Eingaben mit dem Schadcode, die hier notwendig ist – als *Persistent XSS* bezeichnet

[OWASP Foundation e]. Die Speicherung in einer Datenbank führt dazu, dass diese Art von XSS am einfachsten entdeckt werden kann [Hoffman 2024].

3.2.2 Reflected XSS

Reflected-XSS-Angriffe werden so genannt, da der Schadcode direkt zum Client „reflektiert“ wird und – anders als bei Stored XSS – nicht gespeichert werden muss [Hoffman 2024].

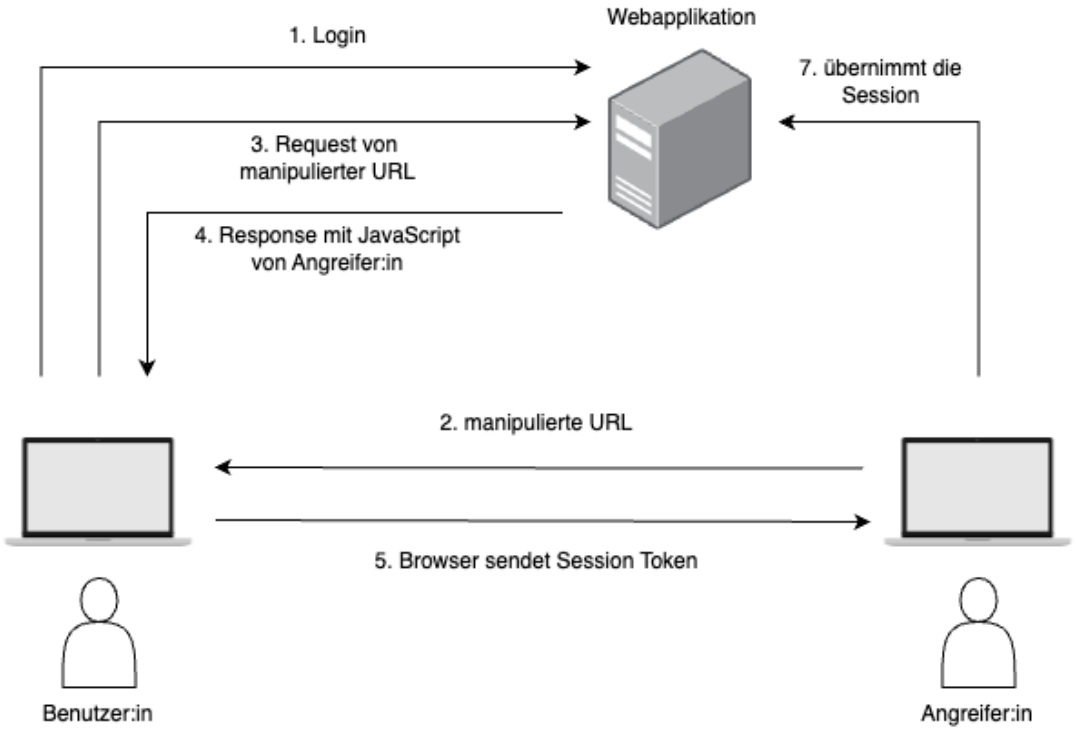

Abbildung 3.4: Reflected-XSS-Angriff, nach [Gebeshuber u. a. 2019, S. 273], Icons: draw.io (CC BY 4.0)

Durch die Verwendung nicht validierter und bereinigter Nutzer:inneneingaben als Teil des HTML wird die Ausführung von HTML und JavaScript im Browser ermöglicht [OWASP Foundation 2017]. Ein:e Angreifer:in kann so eine URL manipulieren und diese einer anderen Person direkt oder indirekt zukommen lassen (Abbildung 3.4, 2.). Wenn eine solche (gezielt gewählte oder zufällige) zweite Person den Link aufruft (3.), wird das Script als Teil der Response übermittelt, ohne dass der Server es verarbeitet (4.), und im

Browser ausgeführt. So kann auch hier etwa ein Session Token an die angreifende Person gesendet (5.). und die aktive Sitzung übernommen werden [Gebeshuber u. a. 2019].

Diese Art wird von der OWASP Foundation auch als *Type I* oder *Non-Persistent XSS* bezeichnet, da die Eingaben mit dem Schadcode für den Angriff nicht permanent gespeichert werden müssen [OWASP Foundation e]. Dies bedeutet, dass Angriffe von der angegriffenen Seite nicht bemerkt werden – es wird nie etwas gespeichert, also kann auch kein Scan das Script finden [Hoffman 2024, S. 124]. Nutzer:innen können direkt angegriffen werden, indem ihnen eine manipulierte URL gesendet wird – es ist aber entsprechend schwieriger als bei Stored XSS, größere Gruppen von Nutzer:innen anzugreifen [Hoffman 2024].

3.2.3 DOM-based XSS

Bei dieser Art des Angriffs wird das Document Object Model (DOM) (siehe Abschnitt 2.3) im Browser des Opfers verändert und dadurch der Schadcode ausgeführt. Dabei wird nicht – wie bei den beiden vorigen Arten – die von der Webseite gesendete HTTP-Response verändert, sondern die Reaktion des Browsers auf diese [OWASP Foundation c]. Es ist dabei keine Interaktion mit dem Server notwendig [Hoffman 2024].

Damit DOM-XSS funktionieren kann, muss die angegriffene Webanwendung sowohl eine *source* – ein DOM-Objekt, das Text speichern kann – als auch eine *sink* – eine DOM-API, die ein in Textform gespeichertes Script ausführen kann – aufweisen [Hoffman 2024]. Über diese Elemente kann ein:e Angreifer:in eine URL manipulieren, bei der etwa eine Suche oder ein Filter nicht in einer Anfrage an den Server resultiert, sondern clientseitig vorgenommen wird, und hier einen Script-Befehl einfügen (Abbildung 3.5, 2.). Wenn eine andere Person die resultierende URL aufruft (3.), wird die URL verarbeitet, um die Such- oder Filterergebnisse anzuzeigen (4.), das Script wird ausgeführt (5.) und etwa das Session Token an die angreifende Person gesendet (6.).

Diese Art des Angriffs wird auch als *Type 0* bezeichnet [OWASP Foundation e]. Ob DOM-based XSS erfolgreich sein kann, hängt vom Browser und der Browserversion ab, was auch die Reproduzierbarkeit und damit die Bekämpfung erschwert [Hoffman 2024].

Eine besonders spezialisierte Form von DOM-XSS, *Mutation-Based XSS (mXSS)*, nutzt die Art, wie Browser die Verarbeitung von DOM-Nodes optimieren, um eine zunächst von Filtern nicht erkennbare Payload nach Passieren der Filter in Schadcode zu mutieren und

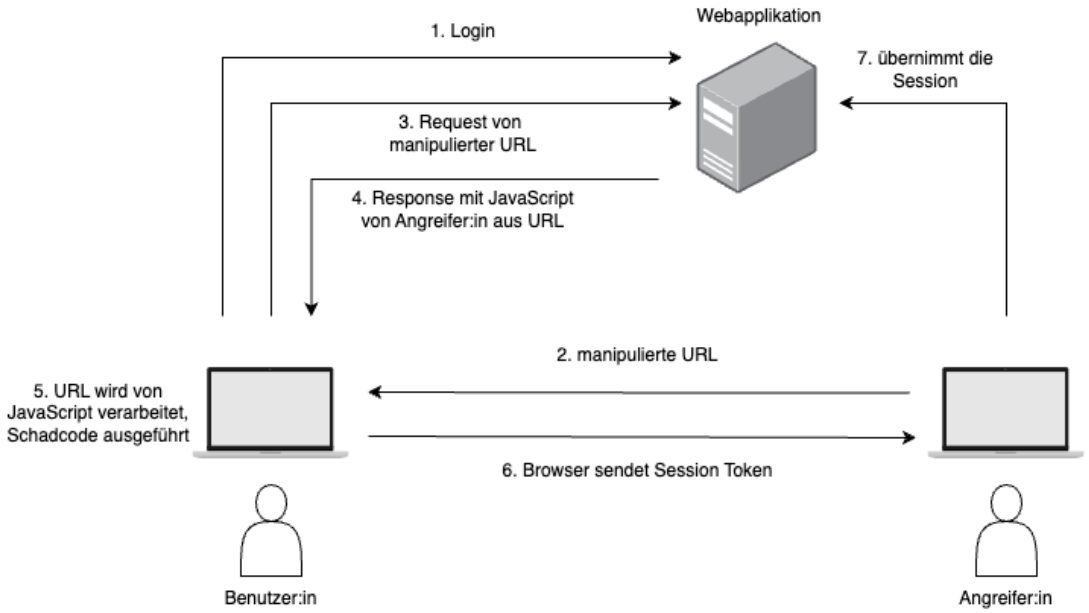

Abbildung 3.5: DOM-Based-XSS-Angriff, nach [Stuttard und Pinto 2011], Icons: draw.io (CC BY 4.0)

ist damit paradoxerweise besonders für gut geschützte Webanwendungen eine Bedrohung [Hoffman 2024].

3.2.4 Server XSS vs. Client XSS

Die verschiedenen Arten von XSS-Angriffen können anstelle der Aufteilung in die drei bereits genannten Typen alternativ nach dem Ort der Nutzung von nicht vertrauenswürdigen Daten als *Client XSS* und *Server XSS* klassifiziert werden. In diesem Modell ist DOM-based XSS eine Teilmenge des Client XSS, während Reflected und Stored XSS sowohl als Server XSS als auch als Client XSS auftreten können [OWASP Foundation e; Hoffman 2024].

3.3 Verhinderung

Die Basis für die Verhinderung von XSS-Attacken ist die strikte Trennung von Daten, die nicht als vertrauenswürdig bekannt sind – also allen Eingaben – von aktiven Browserinhalten [OWASP Foundation 2017]. Gefährliche Zeichen müssen vor der Verwendung solcher Inhalte neutralisiert bzw. maskiert, also ihrer Funktion beraubt werden [Pfleeger u. a. 2023; Amberg und Schmid 2024]. Bevor Daten in eine HTML-Seite eingebettet werden, werden solche Zeichen in ihre HTML-Codes umgewandelt und können damit nicht mehr als JavaScript interpretiert werden [Gebeshuber u. a. 2019]. Wird dies in Kombination mit einer Validierung, dass die Eingaben die richtige Form haben, für alle Variablen durchgeführt, spricht die OWASP-Stiftung von „*perfect injection resistance*“, also einer perfekten Widerstandsfähigkeit gegen Injection-Angriffe [OWASP Foundation a]. Dieses Neutralisieren ist in einigen Webframeworks bereits integriert, etwa seit einigen Jahren in aktuellen Versionen von Ruby on Rails und React JS [OWASP Foundation 2017]. [Gebeshuber u. a. 2019] gehen soweit, zu sagen, in Webframeworks, die fertige Webkomponenten bereitstellen, werde die „HTML-Codierung üblicherweise automatisch vom Framework erledigt“ [Gebeshuber u. a. 2019, S. 276]. Sie heben die Notwendigkeit von Vorsicht bei der eigenen Generierung von HTML-Code hervor. Die OWASP Foundation betont die Notwendigkeit, auch bei der Verwendung von Frameworks deren Grenzen zu kennen und Fälle zu berücksichtigen, die nicht abgedeckt sind [OWASP Foundation 2017, OWASP Foundation a]. Auch außerhalb von Webframeworks stehen Entwickler:innen Bibliotheken für die Bereinigung von HTML (*HTML sanitization*) zur Verfügung, von denen Beispiele für Rust in Unterabschnitt 4.4.2 betrachtet werden.

Gegen Reflected- und Stored-XSS-Attacken hilft die Bereinigung von HTTP-Request-Inhalten auf der Basis des Kontexts [Gebeshuber u. a. 2019, OWASP Foundation 2017].

Bei der Verwendung von Nutzer:inneneingaben innerhalb einer Webanwendung sollte *output encoding* angewandt werden, also die Umwandlung der Eingaben in sichere Formen [OWASP Foundation a]. Hierfür stehen unterschiedliche Bibliotheken zur Verfügung und die OWASP-Stiftung betont die Wichtigkeit, die jeweils richtige Bibliothek für den Verwendungszweck auszuwählen [OWASP Foundation a]. Gegen Stored-XSS-Angriffe kann auch das Scannen von Datenbanken auf gespeicherte Script-Befehle helfen, wobei dies komplexere Angriffe nicht aufspüren kann (z. B. bei base64-kodiertem oder aufgeteiltem Scripttext) [Hoffman 2024].

Als weitere Schicht der Verteidigung können HTTP-Header dienen. [Alcorn u. a. 2014, S. 13] bezeichnen diese als „a natural mechanism for the server to instruct the browser to introduce additional security controls“, da sie dazu dienen, Anweisungen für Request oder Response zu geben. Dazu gehört auch der `Content-Security-Policy`-Header, den der Webserver setzen sollte [Mozilla Corporation a]. Dies ersetzt den zuvor in Chrome und Edge umgesetzten `X-XSS-Protection`-Header, der XSS-Schwachstellen verursachen konnte, statt vor diesen zu schützen [Mozilla Corporation e]. In einer CSP kann definiert werden, welche Ressourcen für den Aufbau einer Webseite verwendet werden dürfen – vor allem, von wo Ressourcen wie Bilder oder Scripts und welche URLs geladen werden dürfen [Mozilla Corporation a]. Auch Beschränkungen bei der Ausführung von JavaScript können darin festgelegt werden [Alcorn u. a. 2014]. Wie eine Juice-Shop-Challenge, die in Unterabschnitt 5.3.1 beschrieben ist, demonstriert, kann eine CSP allerdings unter Umständen umgangen werden. Um zu verhindern, dass Session Token und Cookies einem XSS-Angriff zum Opfer fallen, sollte für diese der `HttpOnly`-HTTP-Header gesetzt werden, sodass clientseitige Scripts nicht auf sie zugreifen können [Gebeshuber u. a. 2019; Alcorn u. a. 2014].

Auch der Ort, an dem Nutzer:inneneingaben in einer Webanwendung verwendet werden, muss für einen Schutz vor Cross-Site Scripting berücksichtigt werden – hier befinden sich *XSS sinks*, von der OWASP-Stiftung definiert als „places where variables are placed into your webpage“ [OWASP Foundation a]. An diesen Stellen sollten als sicher bekannte Varianten verwendet werden, die Eingaben als Text behandeln und nicht ausführen.

Die Behauptung, dass Nutzer:innen kompromittiert werden, weil sie unvorsichtig seien, bezeichnen [Stuttard und Pinto 2011] als Mythos – mit Stored XSS etwa können selbst extrem sicherheitsbewusste Nutzer:innen erfolgreich angegriffen werden, ohne selbst etwas dazu beizutragen. Dennoch kann es gegen Varianten, die darauf basieren, dass Nut-

zer:innen auf einen manipulierten Link klicken (DOM-based oder Reflected XSS), helfen, das Bewusstsein für die Existenz solcher Angriffe zu wecken.

Um Entwickler:innen über geeignete Maßnahmen gegen XSS-Angriffe zu informieren und als Hilfe bei der praktischen Umsetzung, stellt die OWASP-Stiftung ein *cheat sheet*, also einen „Spickzettel“, zur Verfügung [OWASP Foundation a]. Darin betonen die Autor:innen, dass keine Maßnahme für sich allein einen Schutz vor Cross-Site Scripting ermöglichen könne, sondern eine Kombination von Abwehrtechniken notwendig sei [OWASP Foundation a]. Die Wichtigkeit eines Verständnisses, wie verwendete Frameworks XSS verhindern bzw. wo deren Schutz Lücken aufweist, wird ebenfalls herausgestellt [OWASP Foundation a]. Für das oben erwähnte *output encoding* gibt das *cheat sheet* Hinweise, welche Art für welchen Kontext verwendet werden sollte. Auch die Notwendigkeit, aktuelle Versionen von Bibliotheken für *HTML sanitization* zu verwenden, wird hervorgehoben. Als nicht geeignet für den Einsatz gegen Cross-Site Scripting nennt das *cheat sheet* web application firewalls, den alleinigen Einsatz von CSP-Headern sowie den Ansatz einer unspezifischen Validierung und Bereinigung von Eingaben, ohne den Kontext zu beachten.

Obwohl mit den genannten Maßnahmen ein gewisser Schutz vor XSS-Angriffen möglich ist, gestaltet sich dies in der Praxis schwierig, da immer eine Abwägung zwischen erwünschten und unerwünschten Eingaben und darüber hinaus ein gutes Verständnis verwendeter Bausteine wie Frameworks notwendig ist – so folgern [Pfleeger u. a. 2023, S. 289] nach der Vorstellung von Gegenmaßnahmen: „In general, however, blocking the malicious effect of a cross-site scripting attack is a challenge“.

4 Rust

Die Entwicklung von Rust wurde 2006 von Graydon Hoare begonnen und ab 2010 als Projekt von Mozilla Research durchgeführt [Sharma u. a. 2019; Rust Core Team 2020]. Ziel war es, eine Sprache zu entwickeln, die maschinennahe Programmierung auf Browser- oder Betriebssystemebene ermöglicht, aber die Schwächen älterer in der Systemprogrammierung eingesetzter Sprachen wie C oder C++ im Hinblick auf Speichersicherheit und Nebenläufigkeit vermeidet [Sharma u. a. 2019]. Bei dem Versuch, 2009 und 2011 Teile der Browser Engine *gecko*, die Mozillas Webbrowser Firefox zugrunde liegt, moderne CPUs durch Nebenläufigkeit besser nutzen zu lassen, traten große Probleme zutage [Sharma u. a. 2019]. [Sharma u. a. 2019] zufolge lagen diese an den Schwächen von C++ im Hinblick auf die Entwicklung, Wartung und Verständlichkeit nebenläufiger Programme. Nach dem Scheitern der Versuche wurde basierend auf diesen Erfahrungen mit der Entwicklung von *Servo* begonnen, einer neuen Browser Engine, die in Rust geschrieben wurde [Sharma u. a. 2019]. Die Entwicklung von Servo und die Weiterentwicklung von Rust waren über die folgenden Jahre eng verzahnt [Sharma u. a. 2019]. Auch nach der Herausbildung als unabhängiges Projekt mit der Veröffentlichung von Rust 1.0 im Jahr 2015 wurde Rust weiterhin primär von Mozilla unterstützt [Rust Core Team 2020]. 2021 wurde die Rust Foundation gegründet, bei der neben Mozilla auch AWS (Amazon Web Services), Huawei, Google und Microsoft Gründungsmitglieder waren [Williams 2021].

Rust wird als Open-Source-Projekt entwickelt, zu dem Freiwillige auf `https://github.com/rust-lang/rust` beitragen können. Die Sprache wird aktiv weiterentwickelt und alle sechs Wochen wird eine neue *stable version* des Compilers veröffentlicht [Sharma u. a. 2019; Rust Contributors]. Als Gründe, Rust zu verwenden, nennt das Entwicklungsteam auf GitHub *performance*, *reliability* und *productivity* [Rust Contributors]. So sei die Sprache effizient in der Speichernutzung, schnell, speicher- und threadsicher sowie mit einem hilfreichen Compiler und zusätzlichen Werkzeugen für die Softwareentwicklung ausgestattet.

Im Folgenden wird die Idee von Rust als „sicherer Sprache“ vorgestellt und auf die Grundlagen für Rusts Stärken in der Speichersicherheit und sicheren Nebenläufigkeit eingegangen. Dabei werden auch die Grenzen der Sicherheitszusagen durch die Notwendigkeit der Interaktion mit anderen Sprachen und Systemen aufgezeigt, die auch „*unsafe Rust*“ notwendig machen – Rust-Quelltext, der speziell markiert ist und für den nicht die gleichen zugesicherten Eigenschaften gelten wie für regulären Rust-Code. Die Beliebtheit und Verwendung von Rust wird thematisiert und die Eignung von Rust für die Entwicklung von Webanwendungen betrachtet. Abschließend werden einige verbreitete Frameworks und Sicherheitstools für die Webentwicklung in Rust vorgestellt.

4.1 Sichere Sprache

Speichersicherheit (*memory safety*) bedeutet, dass Variablen in einem Programm während der gesamten Laufzeit nicht auf invalide Speicheradressen zeigen können und das Programm nie Zugriff auf Speicherbereiche hat, auf die es nicht zugreifen soll [Sharma u. a. 2019; Flitton 2023]. Um speichersicher zu sein, dürfen Operationen also nie zu undefiniertem Verhalten führen – einem Zustand, der im Compiler nicht vorgesehen ist, weil er eigentlich nicht vorkommen sollte. Aus undefiniertem Verhalten können unvorhersehbare Konsequenzen folgen – teilweise direkt offensichtlich als Programmabsturz, aber in vielen Fällen als schwer nachvollziehbare Fehler, die von Angreifer:innen ausgenutzt werden können [Sharma u. a. 2019; Blandy u. a. 2021]. Rust sichert – sofern kein *unsafe* Rust und keine Bibliotheken in anderen Sprachen verwendet werden, s. u. – zu, dass ein Programm, das erfolgreich kompiliert, nirgendwo undefiniertes Verhalten aufweist, was eine Ursache zahlreicher Sicherheitsprobleme beseitigt [Blandy u. a. 2021]. Der Compiler stellt sicher, dass angreifbare Schwachstellen wie etwa *buffer overflows*, *segmentation faults*, *use after frees* oder *double frees* nicht auftreten können, teilweise in Kombination mit Überprüfungen zur Laufzeit [Flitton 2023; Blandy u. a. 2021]. Rusts Stärken in dieser Hinsicht fassen [Blandy u. a. 2021, S. 41] als „safety is invisible“ zusammen: Die Sicherheit von Rust-Programmen im Hinblick auf *memory safety* und Nebenläufigkeit liegt darin, was die Sprache nicht erlaubt.

Der Begriff, der für die Sicherheitsaspekte von Rust verwendet wird, ist *safety*, von [Eckert 2023] als Funktionssicherheit und die Freiheit von Bedrohungen durch technische Fehler im Programm charakterisiert. Die Sicherheit vor Angriffen (*security*) – von [Eckert 2023, S. 7] als „Abwehr von Bedrohungen, die durch unberechtigte Zugriffe von außen auf die

zu schützenden Güter des IT-Systems geschehen“ definiert – ist hiermit jedoch eng verknüpft, da durch die Vermeidung technischer Fehler weniger Schwachstellen entstehen, die Angreifer:innen nutzen können. Dabei hängen Speichersicherheit und sichere Nebenläufigkeit – von [Klabnik und Nichols 2023, S. 353] „fearless concurrency“ genannt – eng zusammen, da das Typsystem und das Prinzip von *ownership* als Kernkonzepte von Rust die Basis für beides sind [Sharma u. a. 2019; Klabnik und Nichols 2023].

In Rust sind die Werte von Variablen „Eigentum“ der Variablen. Dieses Konzept von *ownership* ist essentiell für die Funktionsweise der Sprache. Werte können „ausgeliehen“ werden (*borrowing*), sind währenddessen aber nicht veränderbar. Verändert werden können sie nur, wenn sie in das Eigentum einer neuen Variablen (generell in einer anderen Funktion oder einem anderen Block) übergehen. Dann kann auf den Wert nicht mehr über die ursprüngliche Variable zugegriffen werden. Dies kann auch temporär geschehen, durch *mutable borrowing* – dabei kann während des „Ausleihens“ der ausgeliehene Wert verändert werden, während auf die gebende Variable währenddessen nicht zugegriffen werden kann [Matsakis 2017; Flitton 2023; Klabnik und Nichols 2023]. Für jede Referenz muss die Information, wie lang ihre Gültigkeitsdauer ist, vorliegen – wenn sie keinem üblichen Muster entspricht, das dem Compiler bekannt ist, muss diese *lifetime* explizit angegeben werden [Klabnik und Nichols 2023]. Die Einhaltung dieser Regeln wird beim Kompilieren vom *borrow checker* überprüft – wenn sie nicht eingehalten werden, kompiliert der Quelltext nicht. Dies führt dazu, dass die Entwicklung von Rust-Programmen auch als „fighting with the borrow checker“ zusammengefasst wird [Klabnik und Nichols]. [Blandy u. a. 2021, S. 131] sehen dies als Vorteil der Sprache und resümieren: „Rust is all about transferring the pain of understanding your program from the future to the present.“

Zentral für die Sicherheit in Bezug auf Speicherzugriffe und Threads, aber auch für die flexible Verwendbarkeit von Rust, ist das Typsystem, das Entwickler:innen sowohl verschiedene Typen für Daten als auch *Generics* und *Traits* zur Verfügung stellt [Blandy u. a. 2021]. *Traits* definieren Verhalten, das ein Typ aufweist, etwa durch Methoden, die er implementiert oder Konstanten, die er besitzt – ähnlich wie beispielsweise Interfaces in Java. Rust ist eine statisch typisierte Sprache, erlaubt aber dank *type inference* das Weglassen expliziter Typinformationen, wenn nur ein Typ vorliegen kann [Blandy u. a. 2021; Sharma u. a. 2019]. Das Konzept von Null existiert nicht (außer in der Interaktion mit anderen Sprachen) [Klabnik und Nichols 2023; Sharma u. a. 2019]. Mit Hilfe des dafür entwickelten Werkzeugs *RustBelt* wurde die Sicherheit des Typsystems 2020 formal verifiziert [Jung 2020]. RustBelt soll auch zukünftig während der Weiterentwicklung von

Rust angewandt werden [Jung 2020]. Der Compiler kann aufgrund der statischen Typisierung Typprüfungen durchführen, da er Informationen über alle Variablen und deren Typen vorliegen hat und Variablen aufgrund der Tatsache, dass Rust *strongly typed* ist, nicht ihren Typ ändern können [Sharma u. a. 2019].

Teilweise muss Quelltext geschrieben werden, der nicht durch die statische Prüfung bei der Kompilierung als sicher bestätigt werden kann, weil dafür nicht genügend Informationen vorliegen – etwa für die Interaktion mit dem Betriebssystem. Für diese Fälle existiert das Konzept von *unsafe*, bei dem mit Hilfe dieses Keywords für den so annotierten Block einige zusätzliche Anweisungen möglich sind, die nicht bei der Kompilierung überprüft werden können [Klabnik und Nichols 2023]. Über das *foreign function interface* (FFI) können auch Bibliotheken in anderen Programmiersprachen verwendet werden, was ebenfalls dazu führt, dass Rusts Compiler seine Garantien für die von der Verwendung betroffenen Teile des Programms nicht geben kann [Lyu 2021].

Indem Rust generell Speichersicherheit und sichere Nebenläufigkeit zusichern kann, sind [Sible und Svoboda 2022] zufolge sieben der *CWE Top 25 Most Dangerous Software Weaknesses* aus dem Jahr 2022 abgedeckt. Sie folgern: „Consequently, Rust developers must remain vigilant for addressing many other kinds of security in Rust.“ [Sible und Svoboda 2022]. [Blandy u. a. 2021, S. 3] stimmen zu, dass Rust zwar zahlreiche Bugs nicht entdecken könne, betonen jedoch: „But in practice, taking undefined behavior off the table substantially changes the character of development for the better“.

4.2 Verwendung und Verbreitung

Rust ist zunächst eine Sprache für *systems programming* und versucht hier, eine bessere Alternative zu C und C++ zu sein [Blandy u. a. 2021]. [Sharma u. a. 2019, S. 9] beschreiben Rust als „a general purpose multi-paradigm language“, die aber auf Systemprogrammierung abziele. Gleichzeitig könne man dank der Ausdrucksstärke der Sprache unter anderem auch hochperformante Webanwendungen schreiben. [Blandy u. a. 2021] zufolge ist neben Sicherheit ein weiteres Ziel von Rust, dass es durch seine Flexibilität und die auch von [Sharma u. a. 2019] gelobte Ausdrucksstärke eine angenehm zu verwendende Sprache sei. Sie sehen die Erfüllung dieses Ziels durch die tatsächliche Verwendung von Rust in vielen verschiedenen Bereichen und einer großen Menge an Programmen bestätigt. Unter Entwickler:innen ist Rust sehr beliebt, so nahm es seit 2016 acht Jahre in Folge den ersten Platz auf der Liste der „most loved“ (bzw. in 2023 „most admired“)

Programmiersprachen des *Stack Overflow Annual Developer Survey* ein [Stack Overflow 2016; Stack Overflow 2017; Stack Overflow 2018; Stack Overflow 2019; Stack Overflow 2020; Stack Overflow 2021; Stack Overflow 2022; Stack Overflow 2023]. Im Jahr 2015, in dem Rust 1.0 veröffentlicht wurde, hatte die neue Sprache bereits Platz drei belegt [Stack Overflow 2015]. Den Grund für die Begeisterung vieler Entwickler:innen für Rust sieht [Lyu 2021] in der Kombination von Eigenschaften, die vorher als unvereinbar galten: Speichersicherheit und Performanz, Produktivität und Kontrolle auf einer niedrigen, maschinennahen Ebene. [Blandy u. a. 2021] schätzen die aktive Community, die guten Tools sowie die Eignung für den Einsatz in großen und komplexen Projekten als wichtige Vorzüge für die Verwendung der Sprache ein.

Eine Besonderheit von Rust ist die Unterstützung bei der korrekten Verwendung von Nebenläufigkeit – einem Gebiet, das [Blandy u. a. 2021, S. 3] zufolge in C und C++ „notoriously difficult to use“ sei, sodass es häufig nur verwendet werde, wenn dies unbedingt notwendig sei. Da Rust durch die integrierte Memory Safety bereits sicherstellt, dass es keine *data races* geben kann, ist die Grundlage für sichere Nebenläufigkeit bereits gegeben. Für Daten, die geteilt und verwendet werden, stellt Rust sicher, dass *synchronization primitives* richtig verwendet werden [Blandy u. a. 2021]. Für komplexere Anwendungszwecke von Nebenläufigkeit stehen Bibliotheken bereit.

Für die Organisation von Quelltext verwendet Rust *crates*, die [Klabnik und Nichols 2023] als „tree of modules that produces a library or executable“ beschreiben. Für die Verwendung solcher Bibliotheken nutzt Rust als *build tool* und Paketmanager *Cargo* [Crichton 2014; Blandy u. a. 2021, S. 4]. Cargo verwaltet auch transitive Abhängigkeiten, die durch verwendete Bibliotheken entstehen [Blandy u. a. 2021, S. 176]. Als zentrale Platform für die Veröffentlichung von Bibliotheken dient *crates.io* (`https://crates.io`), wo auch ältere Versionen weiter bereitgestellt werden [Crichton 2014]. Alternativ können Crates auch aus Git Repositories oder mit Angabe eines lokalen Pfads verwendet werden [Blandy u. a. 2021]. Jede Bibliothek beschreibt sich und die benötigten Dependencies inklusive der kompatiblen Versionen der Dependencies in einer Manifest-Datei, die das Format TOML (Tom's Obvious Minimal Language) verwendet [Cargo Contributors]. Versionen können dabei genau festgelegt werden; standardmäßig verwendet Cargo aber die neueste Version, die mit der genannten kompatibel ist – bei 0.0 nur diese, ab 0.1 alle in der gleichen Serie (etwa für 0.6.1 die neueste Version der Serie, z. B. 0.6.3) und ab Version 1.0 alle Versionen innerhalb der Hauptversion (statt 1.0.2 dann z. B. auch 1.80.1). Rust verwendet dabei von Semantischer Versionierung (*semantic versioning*) adaptierte Regeln, sodass inkompatible Veränderungen (*breaking changes*) nur zwischen Hauptversionen auftreten [Blandy

u. a. 2021, S. 203]. Die Sprache ist durch diese Verwendung eines einfachen zentralen Paketmanagers darauf ausgelegt, ihren Nutzer:innen die Zusammenarbeit zu erleichtern, ähnlich wie es npm für JavaScript oder RubyGems für Ruby tun [Blandy u. a. 2021]. Auch Bibliotheken in anderen Sprachen, etwa C, können dank des *foreign function interface* (FFI) aufgerufen werden. Dies bedeutet jedoch, wie in Abschnitt 4.1 ausgeführt, dass Rust seine Sicherheitsgarantien für diesen Teil des Programms nicht geben kann und führt [Lyu 2021, S. 4] zufolge häufig zu „weird and non idiomatic Rust“.

[Blandy u. a. 2021] berichten, dass ihrer Erfahrung zufolge das Vertrauen, dass die Sprache Fehler aufzeigt, zu ambitionierteren und größeren Projekten ermutige. Auch bei Veränderungen an komplexen Programmen sei es von Vorteil, sich auf die von Rust zugesagte Speichersicherheit verlassen zu können. Rust wird inzwischen von zahlreichen großen Firmen und Organisationen im Produktivbetrieb genutzt, darunter Dropbox, Microsoft, Facebook, npm und Cloudflare [Lyu 2021; Sharma u. a. 2019].

4.3 Rust für Webanwendungen

[Lyu 2021, S. 1] bezeichnet Webprogrammierung als „one of the most exciting fields“ für die Verwendung von Rust. Er argumentiert, Rust passe perfekt zur Entwicklung von Webanwendungen, da es Sicherheit (hier „security“), Nebenläufigkeit und „low-level control“ liefere [Lyu 2021, S. 9]. Auch hier liegt ein großer Vorteil in der in Abschnitt 4.1 angesprochenen Prüfung beim Kompilieren, die zahlreiche Schwachstellen zur Laufzeit verhindern kann [Lyu 2021]. Dies ist besonders aufgrund der in Abschnitt 2.3 aufgezeigten Gefährdung von Webanwendungen interessant.

Mit der in Abschnitt 2.3 angesprochenen Komplexität und Interaktivität moderner Webanwendungen geht die Anforderung an diese Anwendungen einher, zahlreiche zeitgleich stattfindende Zugriffe von Nutzer:innen zu verarbeiten. Effizienz und gut funktionierende Nebenläufigkeit sind hier daher besonders wichtig. Rust kombiniert diese mit der Fähigkeit, maschinennah CPU und Speicher direkt zu kontrollieren und so die Effizienz zu steigern [Lyu 2021].

Eine Herausforderung ist das geringe Alter der Sprache und damit auch die teilweise fehlende Reife von Bibliotheken [Lyu 2021]. Rust hat zwar eine aktive Open-Source-Community, dabei aber auch zahlreiche Bibliotheken, die von Grund auf neu entwickelt wurden, teilweise um mit neuen Funktionen und Konzepten zu experimentieren. Dies

führt dazu, dass eine Reihe von Bibliotheken entweder nicht stabil sind oder nicht mehr gewartet werden [Lyu 2021]. Im Bereich der Webanwendungen kommt hinzu, dass Rust ursprünglich für die Systemprogrammierung entwickelt wurde, wie in Abschnitt 4.2 dargelegt, auch wenn die Unterstützung von netzwerkbasierten Anwendungen zunehmend ausgebaut wird [Blandy u. a. 2021, S. 498]. Dennoch existieren Bibliotheken verschiedener Art sowohl für das Frontend als auch das Backend, von denen im Folgenden einige beliebte kurz charakterisiert werden. Außerdem wird mit *Web Assembly* eine Technologie vorgestellt, die in Frameworks und anderen Bereichen der Rust-Webprogrammierung Anwendung finden kann.

4.3.1 Backend-Frameworks

Die folgenden Backend-Frameworks stellen Funktionalität bereit, um einen Webserver zu entwickeln, der serverseitig HTML ausliefert. Dabei betonen alle drei Frameworks die Stärken von Rust: Schnelligkeit, Sicherheit und Nebenläufigkeit [Actix Contributors; Axum Contributors; Rocket Contributors].

Actix Web
https://actix.rs/
Actix Web ist ein etabliertes, aktiv weiterentwickeltes und beliebtes Web Framework mit 147 Releases seit 2017. Aktuell liegt es in Version 4.8.0 vor und wird in fast 58.000 Github-Repositories verwendet (Stand Juli 2024) [Actix Contributors]. Diese Stabilität und der Reifegrad des Frameworks sind Vorteile für den Einsatz. Mit Hilfe von Actix Web kann aus fertig bereitgestellten Komponenten ein Webserver zusammengestellt werden [Blandy u. a. 2021]. [Baumgartner 2023] kritisiert, dass man – anders als bei Axum (s. u.) – nicht die *Tower*-Middleware verwenden kann, hebt aber die hervorragende Dokumentation und Produktionsreife hervor. Actix Web unterstützt HTTP/1 und HTTP/2 sowie TLS und ist laut dem Entwicklungsteam gut für kleine Services im Produktivbetrieb geeignet [Actix Team]. In den *Tech Empower Web Framework Benchmarks*, einem Vergleich von 159 Web Frameworks im Hinblick auf ihre Schnelligkeit, belegte Actix Web 2023 den 13. Platz [Tech Empower 2023].

Axum
https://github.com/tokio-rs/axum
Axum ist ein modulares Web Framework, das auf *Tokio* (https://tokio.rs) basiert, einer asynchronen Laufzeitumgebung, die auf Rusts `async/await` aufbaut [Lerche].

Seit 2021 wurden 78 Releases veröffentlicht. Axum wird in mehr als 41.000 Github-Repositories verwendet und hat mit Abstand die meisten kürzlichen Dowloads auf crates.io (siehe Tabelle 4.1, Stand Juli 2024) [Axum Contributors]. In den *Tech Empower Web Framework Benchmarks* belegte Axum 2023 den 10. Platz [Tech Empower 2023]. [Baumgartner 2023] lobt die *developer experience* dank der einfach verwendbaren API, die auf Rusts Traits ohne Verwendung von Macros basiert: Durch die Kapselung in Traits sei es es für Entwickler:innen einfach, aus Axums Komponenten Anwendungen zusammenzustellen. Das darunterliegende Tokio ist dem ursprünglichen Entwickler zufolge gut für IO-intensive Anwendungen geeignet, nicht hingegen für rechenintensive Einsatzzwecke oder Anwendungen, die viele Dateien verarbeiten müssen [Lerche]. Ein Risiko beim Einsatz von Axum besteht darin, dass es noch nicht in Version 1.0 veröffentlicht ist und sich daher noch stark verändert [Baumgartner 2023; Axum Contributors].

Rocket

https://rocket.rs/

Rocket ist das älteste der verbreiteten Backend Frameworks und wurde seit 2016 in 58 Releases weiterentwickelt. Es wird in mehr als 30.000 Github-Repositories verwendet und liegt aktuell in Version 0.5.1 vor (Stand Juli 2024) [Rocket Contributors]. Damit ist es allerdings weniger aktiv als Actix Web und Axum und auch die Frequenz der Releases ist geringer. Die Downloadzahlen auf crates.io sind sehr viel niedriger als die der beiden anderen Frameworks (vgl. Tabelle 4.1). Da Rocket noch nicht Version 1.0 erreicht hat, sind größere Änderungen bei Veröffentlichung von neuen Versionen möglich. Rocket versucht, alle Funktionen für die Bereitstellung einer Webanwendung zu bieten, unter anderem mit Eingabeformularen, Unterstützung für die Datenbankanbindung und eigenem Templating. [Baumgartner 2023] beschreibt dies als einen „batteries-included approach“ und betont, dass dies auch bedeute, dass Entwickler:innen sich auf die Art, wie Rocket arbeitet, einstellen müssen [Baumgartner 2023]. Anders als die bereits genannten Frameworks wurde Rocket nicht in den *Tech Empower Web Framework Benchmarks* getestet.

	Actix Web	Axum	Rocket
verfügbar seit	2017	2021	2016
Releases auf crates.io	147	78	58
Version	4.8.0	0.7.5	0.5.1
GitHub-Repositories[a]	58.000	41.500	30.300
gesamte Downloads auf crates.io	21.632.755	53.163.534	4.947.359
kürzliche Downloads auf crates.io[b]	3.010.351	12.928.985	432.699
Macro-Verwendung	ja	nein	ja
Benchmark-Platzierung[c]	13	10	-

[a] gerundet auf Hunderterstelle [b] in den vorigen 90 Tagen
[c] Tech Empower Web Framework Benchmarks
Stand: 2024-07-14

Tabelle 4.1: Backend-Frameworks

4.3.2 WebAssembly (Wasm)

WebAssembly ist kein Framework, sondern ein für Effizienz im Hinblick auf Größe und Schnelligkeit entwickeltes binär codiertes Format, das von Gruppen des *W3C* (*World Wide Web Consortium*) entwickelt wird. In diesen Gruppen sind die am weitesten verbreiteten Webbrowser vertreten und die Verwendung von WebAssembly ist in Firefox, Chrome, Safari und Edge umgesetzt [Mozilla Corporation d]. Wasm bietet eine auf einem Stack basierende virtuelle Maschine, mit deren Hilfe Anweisungen im Browser in beinahe nativer Geschwindigkeit ausgeführt werden sollen, indem Hardwareeigenschaften möglichst gut ausgenutzt werden [Mozilla Corporation d; Lyu 2021]. Rust kann direkt in Wasm kompiliert und dann in Webbrowsern ausgeführt werden und bildet so auch die Basis für Rust-Frontend-Frameworks. [Lyu 2021] hebt hervor, dass dies großes Potential für die Frontendentwicklung für hoch performante Webanwendungen habe. WebAssembly ist als Ergänzung und nicht als Ersatz für JavaScript intendiert [Mozilla Corporation d]. So kann es etwa nicht direkt auf das DOM zugreifen, sondern muss dafür JavaScript nutzen [Mozilla Corporation c].

4.3.3 Frontend- und Full-Stack-Frameworks

Frameworks für die Entwicklung von Web-Frontends in Rust sind noch recht jung: Yew als das älteste weit verbreitete Framework wird seit 2019 entwickelt [Yew Contributors b].

Auch die Nutzung ist noch nicht so weit verbreitet wie die der Backend-Frameworks, so nutzen lediglich rund 11.000 Projekte auf GitHub das beliebteste Frontend-Framework, während das beliebteste Backend-Framework von ca. 58.000 Repositories verwendet wird. Keins der verbreiteten Frameworks hat bisher Version 1.0 erreicht und Yew warnt explizit vor möglichen mit vorigen Versionen inkompatiblen Änderungen [Yew Contributors b] (Stand Juli 2024).

Zwei der Frameworks nutzen das Konzept von *Virtual DOM* (VDOM), das auch im verbreiteten JavaScript-Framework *React* verwendet wird. Dabei werden Veränderungen in einer virtuellen Repräsentation des DOM umgesetzt und das VDOM ruft die DOM-API nur für die Elemente auf, die sich geändert haben, um diese zu aktualisieren (in React *reconcile* genannt) [Lyu 2021].

Yew

https://yew.rs/

Yew ist ein reines Frontend-Framework, das von React und Elm beeinflusst ist [Lyu 2021; Yew Contributors b]. Es ist das mit Abstand meistgenutzte Rust-Framework für die Erstellung von Frontends mit fast viermal so vielen GitHub-Repositories und mehr als doppelt so vielen Downloads auf crates.io wie das zweithäufigste Framework (siehe Tabelle 4.2). Yew wurde für die Erstellung von *single-page applications* entwickelt; *server-side rendering* ist inzwischen möglich, aber bisher experimentell [Yew Contributors a].

Dioxus

https://dioxuslabs.com/

Dioxus kann verwendet werden, um Webanwendungen genauso wie Desktop- oder mobile Anwendungen zu entwickeln [Dioxus Contributors]. Für Webanwendungen bietet es sowohl direktes Rendern in das DOM mit WebAssembly als auch *server-side rendering* mit clientseitigem *rehydration*, bei dem im Browser interaktive Elemente hinzugefügt werden. Dioxus hat ein Kernteam von Vollzeit-Entwickler:innen [Dioxus Contributors].

Leptos

https://github.com/leptos-rs/leptos

Leptos ist ein Full-Stack-Framework, das auch *server-side rendering* und *server-side rendering with rehydration* ermöglicht [Leptos Contributors]. Der größte Unterschied zu Dioxus im Hinblick auf Webanwendungen besteht darin, dass Leptos kein VDOM (virtual DOM) verwendet [Dioxus Contributors; Leptos Contributors].

	Yew	Dioxus	Leptos
verfügbar seit	2019	2022	2023
Releases auf crates.io	38	22	83
Version	0.21.0	0.5.1	0.6.12
GitHub-Repositories[a]	11.800	2.000	3.100
gesamte Downloads auf crates.io	1.227.983	223.839	465.676
kürzliche Downloads auf crates.io[b]	117.132	59.188	125.947
VDOM-Verwendung	ja	ja	nein
client-side rendering	ja	ja	ja
server-side rendering	experimentell	ja	ja

[a] gerundet auf Hunderterstelle [b] in den vorigen 90 Tagen
Stand: 2024-07-14

Tabelle 4.2: Frontend-Frameworks

4.4 Sicherheitstools in Rust

Das Rust-Ökosystem verfügt über eine Reihe von Bibliotheken und Projekten, die verschiedene Sicherheitsaspekte abdecken. Im Folgenden werden einige allgemeine sowie einige speziell gegen XSS einsetzbare Werkzeuge vorgestellt, um ihre Eignung für den möglichen Einsatz im Prototyp bewerten zu können.

4.4.1 Allgemeine Sicherheitswerkzeuge

Die im Folgenden vorgestellten Bibliotheken sind für den Einsatz in allen Arten von Rust-Anwendungen ausgelegt, nicht spezifisch für den Schutz von Webanwendungen vor Cross-Site Scripting.

mir-checker (statische Analyse)
https://github.com/lizhuohua/rust-mir-checker
MirChecker ist ein statisches Analysetool, das auf Rusts *Mid-level Intermediate Representation (MIR)* arbeitet und dort Programmierfehler und Schwachstellen entdecken soll [Li u. a. 2021]. MIR ist ein Zwischenzustand bei der Kompilierung von Rust-Quelltext, bei dem Parsen und Vereinfachung der Syntax sowie Typüberprüfungen stattgefunden haben, *borrow checking* (siehe Abschnitt 4.1) und Optimierung aber noch ausstehen [Matsakis

2016]. MirChecker verwendet noch die Rust-Edition 2018 sowie einige veraltete Versionen von Bibliotheken und scheint nicht mehr aktiv entwickelt zu werden – seit November 2022 gab es keine Änderungen am eigentlichen Tool.

Miri (Erkennung undefinierten Verhaltens)
https://github.com/rust-lang/miri
Auch *Miri* arbeitet auf Rusts MIR. Es ist ein Analysetool, das undefiniertes Verhalten in Rust-Projekten finden soll [Miri Contributors], das durch *unsafe Rust* oder Bibliotheken in anderen Sprachen verursacht werden kann (siehe Abschnitt 4.1). Es kann in eine CI-Pipeline integriert werden, wird aktiv weiterentwickelt und soll auch mit zukünftigen Rust-Versionen kompatibel bleiben [Miri Contributors].

cargo-careful (Erkennung undefinierten Verhaltens)
https://crates.io/crates/cargo-careful
Mit Hilfe von *cargo-careful* kann undefiniertes Verhalten in Rust-Quelltext erkannt werden. Laut der Beschreibung auf `crates.io` erkennt die Bibliothek dabei weniger als Miri (s. o.), kann aber auch bei der Verwendung anderssprachiger Bibliotheken verwendet werden und ist schneller [Cargo-careful Contributors a]. Die Bibliothek wird aktiv weiterentwickelt [Cargo-careful Contributors b].

sanitizer (Bereinigung von *Structs*)
https://crates.io/crates/sanitizer
Die Bibliothek *Sanitizer* bietet Methoden und ein Macro, um Felder von Rust-*Structs* zu bereinigen, sowie die Möglichkeit, eigene Funktionen dafür zu definieren.

validator (Input-Validierung)
https://crates.io/crates/validator
Validator bietet die Möglichkeit, verbreitete Inputtypen zu validieren (u. a. E-Mails, URLs, Kreditkartennummern) sowie eigene Validierungsfunktionen zu definieren.

4.4.2 XSS-spezifische Werkzeuge

Für den Schutz von Webanwendungen vor XSS existieren eigene Bibliotheken, die sich auf das HTML sowie auf HTTP-Header konzentrieren.

libinjection-rs (XSS-Erkennung)
https://crates.io/crates/libinjection
Libinjection-rs ist die Rust-Anbindung für eine C/C++-Bibliothek, die primär SQL analysiert, aber auch XSS-Erkennung ermöglicht [LibInjection Contributors; LibInjection-Rs Contributors]. Es wird nur erkannt, ob eine bekannte Form von Cross-Site Scripting vorliegt, eventuell vorhandene Script-Befehle werden aber nicht entfernt. Dies entspricht der Funktionsweise von web application firewalls, deren Einsatz die OWASP-Stiftung nicht empfiehlt: Sie können nur gegen einen kleinen Anteil von Angriffen helfen, es werden häufig Möglichkeiten, sie zu umgehen, entdeckt und sie können DOM-based XSS nicht erkennen [OWASP Foundation a].

sanitize_html (HTML sanitization)
https://crates.io/crates/sanitize_html
Sanitize-html ist eine Bibliothek zur Bereinigung von HTML von potentiell gefährlichen Elementen wie Script-Tags (`<script>...</script>`) oder URLs [Sanitize_-html Contributors].

ammonia (HTML sanitization)
https://crates.io/crates/ammonia
Ammonia ist eine Bibliothek für *HTML sanitization*, also die Bereinigung von HTML durch Entfernen von Elementen, die gefährlich sein könnten. Das Tool baut ein DOM auf, wie ein Webbrowser dies tun würde. Unerwünschte Elemente, die es beim Traversieren des DOM findet, ersetzt es [Ammonia Contributors]. Dabei verwendet es eine Whitelist und einen im Mozilla-Servo-Projekt entwickelten HTML-Parser, der in Rust geschrieben ist [Ammonia Contributors].

armor (HTTP-Header)
https://crates.io/crates/armor
Armor setzt verschiedene HTTP-Header, darunter `X-XSS-Protection` [Armor Contributors 2020]. Da das Setzen dieses Headers XSS ermöglichen kann statt es zu verhindern, wie in Abschnitt 3.3 erwähnt, ist die Verwendung nicht ratsam. Armor unterstützt auch das Setzen einer CSP, wird aber nicht mehr weiterentwickelt, da seine Funktionalität in die `http-types`-Bibliothek integriert wurde [Armor Contributors 2020]. Auf crates.io steht Armor weiterhin zur Verfügung (Stand 2024-07-22).

http-types (HTTP-Header)
https://crates.io/crates/http-types
Http-Types bietet mit dem Modul `security` die Möglichkeit, HTTP-Header zu setzen. Diese umfassen neben dem nicht zu empfehlenden `X-XSS-Protection`-Header (s. o.) auch den gegen XSS-Angriffe einsetzbaren `Content-Security-Policy`-Header [Http-Types Contributors].

5 Trainings- und Demonstrationsanwendung OWASP Juice Shop

Der *OWASP Juice Shop*, im Folgenden auch kurz Juice Shop genannt, ist eine absichtlich mit zahlreichen Sicherheitsschwachstellen ausgestattete Webanwendung, die zu Trainings- und Demonstrationszwecken entwickelt wurde [Kimminich]. Der ursprüngliche Entwickler, Björn Kimminich, beschreibt es als das Gegenteil einer *best-practice*-Anwendung – wie auch der Name, der auf dem deutschen Begriff *Saftladen* basiert, nahelegt [Kimminich]. Auf GitHub beschreibt das Juice-Shop-Team sein Projekt kurz als „[p]robably the most modern and sophisticated insecure web application“ [Juice Shop Contributors].

Ursprünglich für die firmeninterne Weiterbildung entwickelt, wurde Juice Shop 2016 als *OWASP Tool Project* der OWASP-Stiftung (siehe Abschnitt 2.3) akzeptiert und ist seit 2018 eins von aktuell fünfzehn *OWASP Flagship Projects* (Stand 2024-06-28) [Kimminich; OWASP Foundation d]. Der Quelltext des von Freiwilligen getragenen Open-Source-Projekts unter MIT-Lizenz kann unter https://github.com/juice-shop/juice-shop heruntergeladen werden. [Kofler u. a. 2023] zufolge ist eine Besonderheit des Juice Shop im Vergleich zu ähnlichen Projekten, dass er gut dokumentiert ist und aktiv weiterentwickelt bzw. gewartet wird.

Die Beschreibungen in diesem Abschnitt basieren auf der Interaktion mit der Webanwendung, dem Quelltext der Anwendung sowie dem Handbuch von Björn Kimminich [Kimminich].

5.1 Funktionsweise

Der Juice Shop verfügt – abgesehen von der Zahlungsabwicklung – über alle Funktionen, die für einen Webshop typisch sind, sowie einige zusätzliche: Produkte mit Reviews, einen Warenkorb mit Artikeln und Liefermöglichkeiten, eine Loginfunktion sowie einen Kund:innenbereich mit Adressen, Passwort, Sicherheitsfragen, Bestellungen, Zahlungsmöglichkeiten, einer Recyclingoption für Verpackungsmaterial und einer digitalen Geldbörse.

Integriert in diesen Shop sind Challenges, bei denen Nutzer:innen absichtlich vorhandene Schwachstellen ausnutzen sollen – der eigentliche inhaltliche Kern der Anwendung. Dabei wird Wert darauf gelegt, dass die Schwachstellen realitätsnah sind [Kimminich].

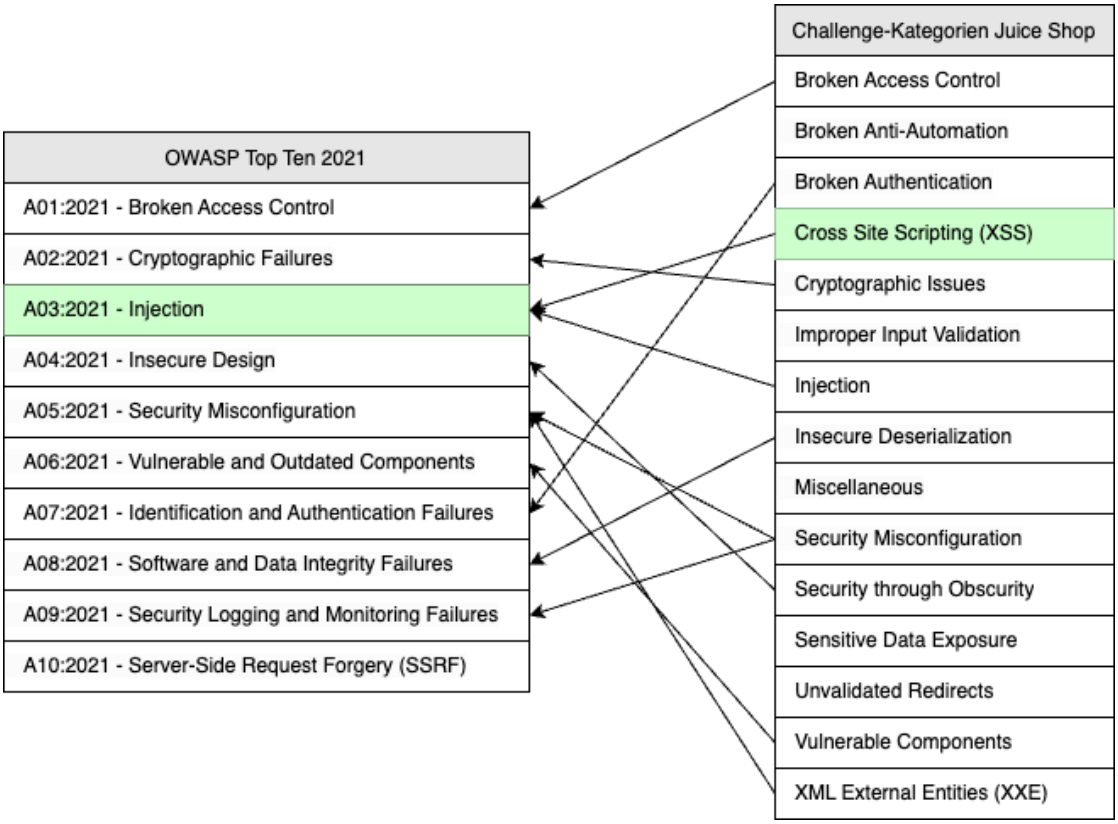

Abbildung 5.1: Mapping OWASP Top Ten 2021 und Kategorien von Juice-Shop-Challenges (alphabetisch geordnet), nach [Kimminich], Template: draw.io (CC BY 4.0)

Die Kategorien entsprechen größtenteils den OWASP Top Ten von 2021 (siehe Abbildung 5.1). Es sind dabei alle Schwachstellengruppen der Top-Ten-Liste abgedeckt, außer Server-side Request Forgery (SSRF). XSS ist, wie in den OWASP Top Ten 2017 und anders als in der Liste von 2021, eine eigene Kategorie mit neun Challenges (Stand: v17.0.0), die in Abschnitt 5.3 näher betrachtet wird.

Ergänzt werden die Challenge-Kategorien durch Schwachstellen, die zu älteren OWASP-Top-Ten-Kategorien gehören (Sensitive Data Exposure, Unvalidated Redirects), Schwachstellen aus den *OWASP API Security Top Ten* (Improper Input Validation) und aus den verwandten OWASP-Listen *OWASP Automated Threats to Web Applications* (Broken Anti-Automation) sowie *OWASP Top 10 Privacy Risks* (Miscellaneous) [Kimminich; OWASP Foundation 2023; OWASP Foundation 2021a].

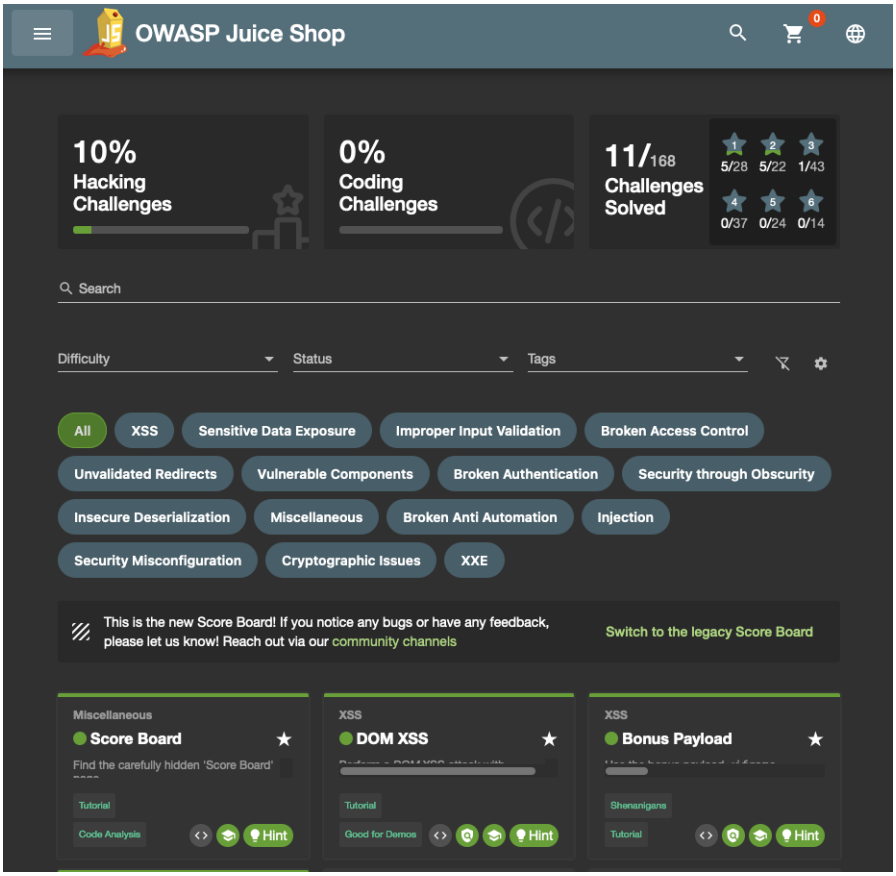

Abbildung 5.2: Screenshot: Score Board, 2024-06-20

Damit Nutzer:innen den eigenen Fortschritt nachvollziehen und Challenges finden können, wird ein Score Board verwendet (das zunächst gefunden werden muss). Jede Challen-

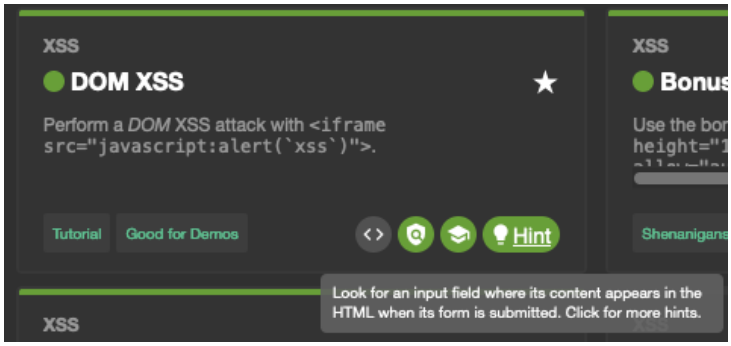

Abbildung 5.3: Screenshot: Challenge, 2024-06-20

ge hat dort eine Kurzbeschreibung mit mindestens einer Schwierigkeitseinstufung (einfach bis sehr schwer: ⋆ bis ⋆⋆⋆⋆⋆⋆). Zusätzlich gibt es einen Hinweis unter „Hint“ (siehe Abbildung 5.3): Bewegt sich der Cursor darüber, wird einem Tooltip ein Kurztext angezeigt, während ein Klick darauf zum entsprechenden Abschnitt im Handbuch *Pwning OWASP Juice Shop* [Kimminich] führt. Einige Challenges verfügen zusätzlich über einen Link zu Material, das über die Verhinderung ähnlicher Schwachstellen informiert (Icon in Form eines Schilds mit Lupe in Abbildung 5.3) oder ein interaktives Tutorial (Gelehrtenhut-Icon).

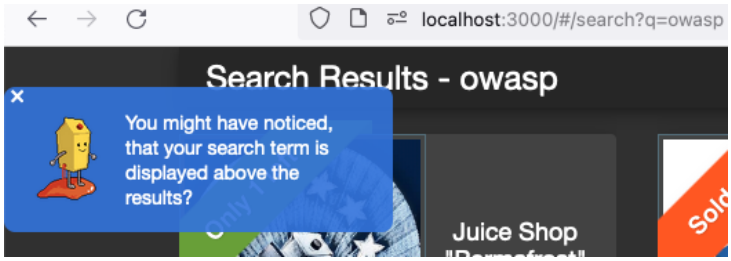

Abbildung 5.4: Screenshot: Tutorial, 2024-06-20

Startet man ein solches Tutorial, dann erscheint in der Bildschirmecke ein erster Hinweis (siehe Abbildung 5.4). Folgt man den enthaltenen Anweisungen, so erhält man schrittweise eine Reihe weiterführender Hinweise, die zur Lösung leiten.

Hat ein:e Nutzer:in eine Challenge gelöst, erscheint eine Benachrichtigung mit der Information (siehe Abbildung 5.5) und die Challenge erhält im Scoreboard einen grünen Balken (siehe Abbildung 5.2).

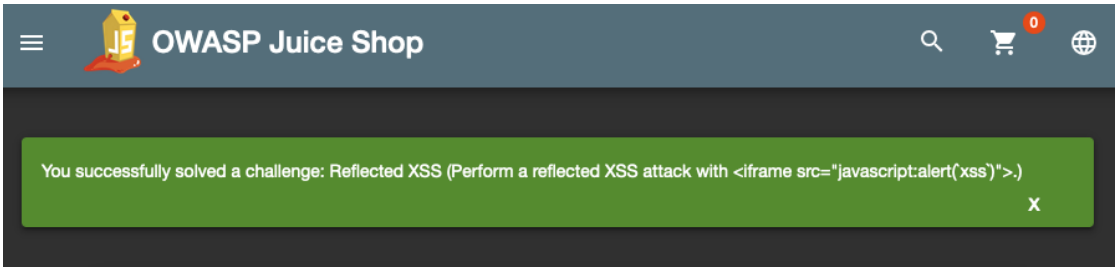

Abbildung 5.5: Screenshot: Benachrichtigung zur Information über eine erfolgreich gelöste Challenge, 2024-06-20

5.2 Architektur und Implementierung

Der Juice Shop ist eine JavaScript-Anwendung, die in einem Webbrowser aufgerufen wird. Dabei wird *TypeScript* verwendet, das JavaScript ein Typsystem hinzufügt [Microsoft Corporation]. Konzeptionell gibt es zwei Domänen: den Shop, der die anzugreifende Anwendung darstellt, und die Challenges, die den Trainingsinhalt bereitstellen. Beide sind sowohl im Frontend als auch im Backend abgebildet (siehe Komponentendiagramm in Abbildung 5.6).

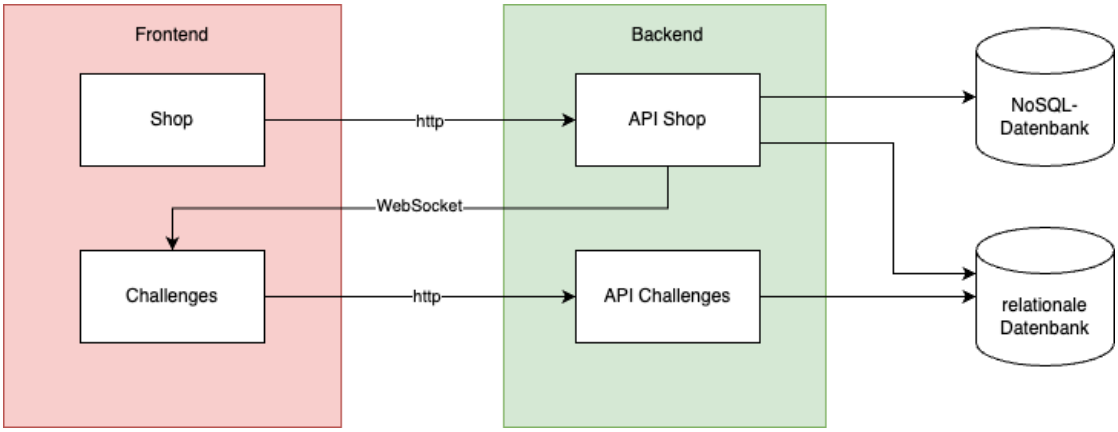

Abbildung 5.6: Komponentenübersicht Juice Shop, Grafiktemplate: draw.io (CC BY 4.0)

Persistiert werden Shopdaten teilweise in einer NoSQL-Datenbank (MongoDB, siehe Abbildung 5.7) (Reviews und Bestellungen), teilweise in einer SQL-Datenbank (SQLite, siehe Abbildung 5.7), während Daten für die Challenges nur in der SQL-Datenbank gespeichert werden. Um statische Daten zur Verfügung zu stellen, werden YAML-Dateien verwendet (für Challenges, Versandarten und Sicherheitsfragen). Auch für die beim Start

schon existierenden Nutzer:innendaten wird eine YAML-Datei verwendet („Content Folder “ in Abbildung 5.7).

Für die Kommunikation zwischen Shop-Frontend und Shop-API sowie zwischen Challenge-Frontend und Challenge-API wird HTTP verwendet (siehe Abbildung 5.6), wobei *JSON over HTTP* genutzt wird. Mit Hilfe von WebSockets informiert das Backend das Frontend, wenn eine Aufgabe gelöst wurde.

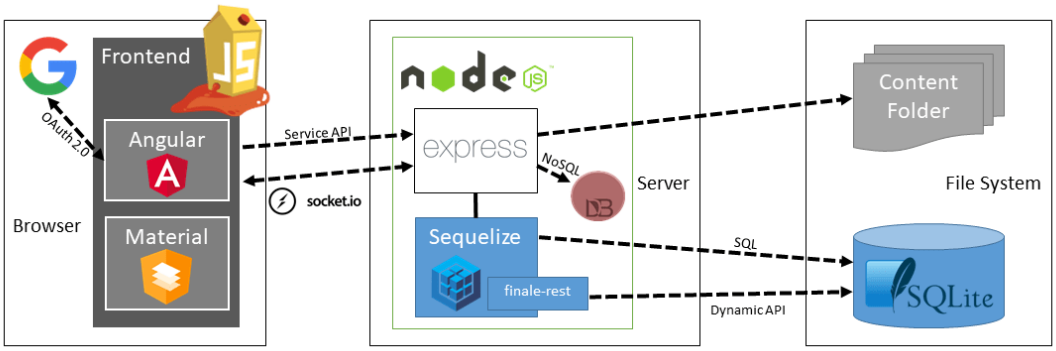

Abbildung 5.7: Architekturübersicht Juice Shop mit verwendeten Technologien [Kimminich]

Es besteht in der Implementierung keine Trennung zwischen den inhaltlich distinkten Komponenten *Webshop* und *Challenges*, wie in Abbildung 5.8 beispielhaft zu sehen ist: In der Funktion für das Hochladen von Zipdateien in `fileUpload.ts` wird auch die Lösungsbedingung für die mit dieser Funktion verknüpften Challenge geprüft und gegebenenfalls die Challenge als gelöst markiert. Dafür wird `challengeUtils` im `lib`-Ordner verwendet, das auch das Senden einer Nachricht wie in Abbildung 5.5 auslöst.

Abbildung 5.8: Screenshot des Juice-Shop-Quelltexts, 2024-06-21

5.3 Analyse der Angriffspfade der XSS-Challenges

Die Juice-Shop-Challenges bilden alle drei Grundarten von XSS (siehe Abschnitt 3.2) ab: Es gibt sechs Challenges zu Stored XSS (im Juice Shop als *Persisted XSS* bezeichnet), eine zu Reflected XSS und zwei zu DOM-based XSS (Stand 2024-06-28, v17.0.0). Die Schwierigkeit der Challenges reicht von einfach (⋆) bis sehr schwer (⋆⋆⋆⋆⋆⋆). Im Folgenden werden diese Challenges kurz vorgestellt.

5.3.1 Stored-XSS-Challenges

API-only XSS (⋆⋆⋆): Mit einem HTML-Request wird der Produkt-Endpoint der Shop-API, der in der grafischen Oberfläche nicht verwendet, aber durch die API bereitgestellt wird, direkt angesprochen. Dabei wird als *Bearer Token* der `Authorization`-Header aus einem zuvor mit dem Browser ausgeführten HTTP-Request eingefügt. Ein Bearer Token ermöglicht in HTTP-Requests der Person, die in seinem Besitz ist (englisch *bearer*), den Zugriff auf Ressourcen, ohne einen kryptografischen Schlüssel einsetzen zu müssen [Jones und Hardt 2012]. Als Produktbeschreibung wird dabei der Script-Befehl `<ifra mesrc="javascript:alert('xss')">` angegeben, der mit den anderen Angaben

gespeichert wird. Ruft man nun alle Produkte auf, wird der Script-Befehl ausgeführt und Inline-Frame sowie Benachrichtigungsdialog erscheinen.

Client-side XSS Protection(⋆⋆⋆): Beim Anlegen eines neuen Benutzer:innenaccounts wird durch clientseitige Eingabenvalidierung verhindert, dass statt der E-Mail-Adresse ein Script gespeichert wird. Dies kann umgangen werden, indem direkt ein POST-Request an den API-Endpunkt gesendet wird, der als E-Mail-Adresse `<iframesrc="javascript:alert('xss')">` enthält. Ruft man anschließend als Admin eingeloggt das Administrationsdashboard auf, wird das gespeicherte Script ausgeführt – ein Benachrichtigungsdialog erscheint und der Inline-Frame ist in der Liste der Nutzer:innen sichtbar.

HTTP-Header XSS (⋆⋆⋆⋆): Juice Shop verwendet einen API-Endpunkt, um die IP-Adresse einer authentifizierten Person zu speichern. Dafür wird statt des Standard-HTTP-Header `X-Forwarded-For` der eigene `True-Client-IP`-Header verwendet. Indem dieser Header im Request durch einen Script-Befehl ausgetauscht wird, wird statt der letzten IP-Adresse das Script aufgerufen, wenn die letzte Login-IP-Adresse angezeigt werden soll. Es wird offenbar nicht validiert, dass tatsächlich das Format einer IP-Adresse vorliegt.

Server-side XSS Protection (⋆⋆⋆⋆): Eine Schwäche in der von Juice Shop verwendeten Bibliotheksversion zur Bereinigung von Eingaben wird ausgenutzt: Diese arbeitet nicht rekursiv, weshalb ein Script-Befehl in einen Kund:innenkommentar eingeschleust werden kann, indem ein weiterer `script`-Tag darin eingefügt wird: Statt `<iframesrc="javascript:alert('xss')">` wird `<{}<script>Foo</script>iframesrc="javascript:alert('xss')">` als Payload verwendet. Das Script wird gespeichert und lässt eine Benachrichtigung erscheinen, wenn es über die Kund:innenfeedback-Slideshow auf der *About*-Seite oder über die Kund:innenfeedbacktabelle im Admin-Dashboard aufgerufen wird.

CSP-Bypass (⋆⋆⋆⋆): Beim Setzen eines Benutzer:innennamens werden die Eingaben bereinigt. Allerdings lässt sich dies einfach umgehen, indem beobachtet wird, was das Ergebnis bei der Eingabe `<iframesrc="javascript:alert('xss')">` ist. Gibt man nun `<<a|ascript>alert('xss')</script>` ein, wird dies gespeichert, aber nicht ausgeführt, da die in Abschnitt 3.3 angesprochene Content Security Policy (CSP) dies verhindert. Um dies zu umgehen, wird die Beobachtung genutzt, dass der CSP-Header die URL zu einem verlinkten Profilbild enthält. Statt einer validen URL zu einem Bild wird eine nicht existierende angegeben sowie eigene Angaben für eine CSP (`script-`

`src 'unsafe-inline' 'self' 'unsafe-eval'`), die die ursprüngliche Content Security Policy überschreiben.

Video XSS (⋆⋆⋆⋆⋆⋆): Eine Schwachstelle beim Hochladen von Zipdateien aus einer anderen Challenge wird genutzt, um einen Script-Befehl in die Untertitel eines Videos einzuschleusen, das bei Aufruf des Videos ausgeführt wird.

5.3.2 Reflected-XSS-Challenge

Reflected XSS (⋆⋆): Der Angriff in dieser Challenge nutzt aus, dass bei der Sendungsverfolgung die Bestellnummer in die URL integriert ist. Die URL wird manipuliert, indem stattdessen `<iframesrc="javascript:alert('xss')">` eingefügt wird. Dies führt dazu, dass beim Aufbau der Seite das Script ausgeführt wird und ein Benachrichtigungsdialog erscheint.

5.3.3 DOM-based-XSS-Challenges

DOM XSS (⋆): Das in Unterabschnitt 3.2.3 angesprochene Muster, bei dem Suchanfragen in die URL eingefügt, clientseitig ausgelesen und verwendet werden, wird hier ausgenutzt, indem `<iframesrc="javascript:alert('xss')">` in das Suchfeld eingegeben wird. Dies führt dazu, dass der Inline-Frame in das DOM eingefügt und das JavaScript ausgeführt wird. Dies lässt einen Benachrichtigungsdialog erscheinen, in dem „xss“ steht.

Bonus Payload (⋆): Diese Challenge funktioniert wie die vorige, fügt aber eine komplexere Payload ein: Der Inline-Frame enthält einen Link zu einer Musikdatei mit einem Lied über den Juice Shop, das automatisch abgespielt wird.

6 Prototyp: eine angreifbare Webanwendung in Rust

Mit seinem Fokus auf Sicherheit und seinem starken Typsystem ist es denkbar, dass Rust gegenüber Angriffen, bei denen andersartiger Input als der geplante übergeben wird, Stärken hat. Zu vermuten ist allerdings, dass bei XSS-Angriffen das Typsystem generell nicht helfen kann, da in vielen Formularen mit Nutzer:inneneingaben notwendigerweise Strings verwendet werden, um Texte abzubilden, die sehr verschieden sein können. Um die Angreifbarkeit von Rust-Webanwendungen exemplarisch zu untersuchen, soll ein Prototyp einer angreifbaren Webanwendung in Rust entwickelt werden, der dem Juice Shop im Kleinen nachempfunden ist. Um die Eigenschaften der Sprache beobachten zu können, sollen dabei, wo möglich, in Rust geschriebene Tools verwendet werden – etwa als Framework oder für den Datenbankzugriff.

Die Kernaufgabe des Systems ist die Bereitstellung eines mit XSS angreifbaren Webshops; die wichtigsten Aspekte der Fachdomäne sind *Produkt* und *Kund:in*. Es gehört zur Kategorie der *Interaktiven Online-Systeme*, die nach [Starke 2018] dadurch gekennzeichnet sind, dass sie in ihre Benutzeroberfläche integrierte Operationen auf Daten enthalten (hier etwa Registrierung und Login von Nutzer:innen), bei denen die Transaktionen festgelegt sind.

Die Mehrzahl der in Abschnitt 5.3 beschriebenen Challenges, die im Juice Shop verwendet werden, um XSS-Angriffe zu demonstrieren, sind auf eine Rust-Webanwendung übertragbar, da sie sich allgemeine Funktionsweisen von Webanwendungen zunutze machen. Ausnahmen bestehen aus einer Challenge, die auf einer speziellen Bibliotheksversion beruht, einer weiteren Challenge, die eine andere Schwachstelle in Juice Shop voraussetzt, sowie zwei Challenges, deren Umsetzung aufwändig wäre, ohne einen zusätzlichen Erkenntnisgewinn zu ermöglichen.

Da bei der Juice-Shop-Challenge *Server-side XSS Protection* aus der Kategorie Stored XSS eine Schwäche der zur Bereinigung von Nutzer:inneneingaben verwendeten Biblio-

Kategorie	Challenge	umsetzbar
Stored XSS	API-only XSS	ja
Stored XSS	Client-side XSS Protection	ja
Stored XSS	HTTP-Header XSS	ja
Stored XSS	Server-side XSS Protection	nein
Stored XSS	CSP-Bypass	aufwändig / unnötig
Stored XSS	Video XSS	nein
Reflected XSS	Reflected XSS	ja
DOM-based XSS	DOM XSS	ja
DOM-based XSS	Bonus Payload	aufwändig / unnötig

Tabelle 6.1: Challenges

theksversion ausgenutzt wird, ist die direkte Übertragung in Rust nicht möglich. Ähnliche Schwächen in Rust-Bibliotheken zu finden und auszunutzen, wäre möglich und interessant (so es diese gibt), führte hier aber zu weit.

Die Challenge *Video XSS* aus der Kategorie Stored XSS nutzt eine Schwachstelle aus einem anderen thematischen Bereich des Juice Shop. Der Angriff selbst ist komplex und im Juice Shop mit der höchsten Schwierigkeitsstufe bewertet. Dies macht die Umsetzung im Rahmen dieser Arbeit unrealistisch.

Die *CSP-Bypass*-Challenge basiert auf einer schlecht eingerichteten Content Security Policy (CSP). Dies ist inhaltlich zwar relevant für Cross-Site Scripting, die Funktionsweise des XSS weicht aber nicht von derjenigen anderer Stored-XSS-Angriffe ab. Aufgrund des Aufwands und des nicht zu erwartenden Erkenntnisgewinns wird diese Challenge nicht umgesetzt.

Die DOM-based-XSS-Challenge *Bonus Payload* unterscheidet sich in ihrer Funktionsweise nicht von der Challenge *DOM XSS* – sie verwendet nur eine Audiodatei als Payload im Sinne eines *easter egg*. Die Umsetzung wäre daher nur sinnvoll, wenn ein eigenes *easter egg* versteckt werden sollte.

Zusätzlich wird im Prototyp eine simple Version einer Stored-XSS-Challenge als Minimalbeispiel umgesetzt, da alle Challenges dieser Kategorie im Juice Shop ein weiteres Element zusätzlich zum eigentlichen Stored-XSS-Angriff enthalten.

Bei der *DOM-XSS*-Challenge ist abzusehen, dass sie nicht mit dem Rust-Teil der Anwendung interagiert, da sie komplett auf der Ebene des Document Object Model (DOM) operiert. Um in der angreifbaren Webanwendung alle Kategorien abzubilden, wird sie dennoch umgesetzt.

6.1 Anforderungen

Der Prototyp soll aus einer, soweit möglich, komplett in Rust entwickelten Webanwendung bestehen. Sie soll einen Shop umsetzen, um dem Vorbild des Juice Shop zu entsprechen. Aus den Angriffen, die abgebildet werden sollen, ergeben sich weitere Anforderungen an die Anwendung.

Kategorie	Challenge	zusätzliche Anforderungen
Stored XSS	Stored XSS[a]	Speicherung von Nutzer:inneneingaben
Stored XSS	API-only XSS	Registrierung und Login, Session Token
Stored XSS	Client-side XSS Protection	clientseitige Validierung von Eingaben
Stored XSS	HTTP-Header XSS	eigener HTTP-Header für Client-IP-Adresse
Reflected XSS	Reflected XSS	manipulierbare URL
DOM-based XSS	DOM XSS	Suchanfragen in URL

[a] eigenes Minimalbeispiel

Tabelle 6.2: Challenges: Anforderungen

Die aus der Fragestellung und dem Vorbild abgeleiteten Anforderungen werden im Folgenden benannt und beschrieben. Auf ihrer Grundlage wurde das Design des Prototyps entwickelt.

ID	Anforderung	Beschreibung
01	Webanwendung in Rust	Die Entwicklung soll komplett in Rust geschehen, wo dies möglich ist.
02	Framework in Rust	Ein Webframework soll neben seiner Eignung auch danach ausgewählt werden, dass es möglichst komplett in Rust umgesetzt ist.
03	Bibliotheken in Rust	Alle verwendeten Bibliotheken sollen Rust-Bibliotheken sein und nicht etwa C-Bibliotheken, deren Verwendung ebenfalls möglich wäre (vgl. Abschnitt 4.1).
04	Shop	In der Webanwendung sollen Produkte wie in einem Shop präsentiert werden.
05	Nutzer:innen-Input	Es soll eine Möglichkeit zur Eingabe von Text durch Nutzer:innen bestehen.
06	Speicherung und Abruf von Nutzer:inneneingaben	Die Speicherung der Eingaben aus Anforderung 05 soll möglich sein. Die gespeicherten Eingaben sollen abgerufen werden können.
07	Registrierung und Login	Nutzer:innen sollen sich registrieren und einloggen können, wobei ein Session Token o. ä. verwendet werden soll.
08	clientseitige Validierung von Eingaben	Um das Umgehen einer clientseitigen Eingabevalidierung zu ermöglichen, soll eine solche umgesetzt werden.
09	eigener HTTP-Header für Client-IP-Adresse	Ein eigener HTTP-Header soll vom Server gelesen werden, um hier das Einfügen eines Script-Befehls zu erlauben.
10	manipulierbare URL	Ein Teil der Anwendung soll aus einer Seite mit einer manipulierbaren URL bestehen.
11	Suchanfragen in URL	Eine Suche soll auf der Ebene des DOM arbeiten, um Suchanfragen in der URL zu verwenden.

Tabelle 6.3: Anforderungen

Ein großer Bestandteil des Juice Shop – neben dem angreifbaren „Shop“ – ist die Begleitung der Challenges durch Tutorials mit Hinweisen, Erfolgsmeldungen bei erfolgreicher Lösung einer Challenge und ein Score Board zum Nachvollziehen des Fortschritts. Aufgrund des Umfangs und der Fragestellung wird im Prototyp darauf verzichtet. Der Erfolg des Angriffs muss selbst erkannt werden. Wird der auch im Juice Shop genutzte Script-

Befehl verwendet, bei dem eine Meldung mit dem Text „xss“ erscheint, ist der Erfolg sofort offensichtlich. Das Hinzufügen der Tutorial-Anteile zum Prototyp zu einem späteren Zeitpunkt ist möglich.

6.2 Komponenten

Das Komponentendesign orientiert sich am Juice Shop, wobei durch den Verzicht auf die Nachverfolgung des Challengefortschritts bzw. der Lösung die Komponenten für diese Teile zunächst entfallen, sowohl im Frontend als auch im Backend (in Abbildung 6.1 ausgegraut). Diese könnten in einem nächsten Schritt ergänzt werden. Der Prototyp besteht somit aus einer Shop-Komponente im Frontend und einer Shop-API im Backend, die über HTTP kommunizieren und ein Client-Server-Modell abbilden (vgl. [Starke 2018]). Für diesen Anwendungsfall, bei dem weder hohe Benutzer:innenzahlen oder große Datenvolumen noch die Notwendigkeit besonders guter Skalierbarkeit zu erwarten sind, bringt eine NoSQL-Datenbank keine Vorteile. Daher wird, anders als im Juice Shop, nur eine relationale Datenbank statt der dort verwendeten Kombination aus einer relationalen und einer NoSQL-Datenbank eingesetzt. Für die hier notwendigen kurzen Transaktionen und Datensätze mit fester Struktur ist eine relationale Datenbank gut geeignet (vgl. [Starke 2018].

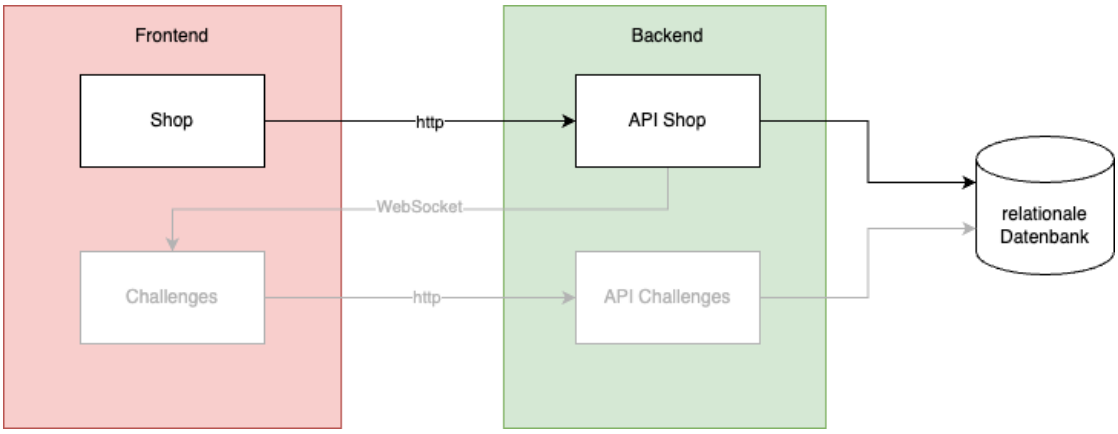

Abbildung 6.1: Komponentenübersicht Prototyp, Grafiktemplate: draw.io (CC BY 4.0)

Der Prototyp ist in die Ebenen *Präsentation* (Shop-Frontend), *Fachdomäne* (Shop-Backend) und *Infrastruktur* (Persistenz und Datenhaltung) aufgeteilt (vgl. [Starke 2018]). Dabei wird eine RESTful-Architektur verwendet: Der Webbrowser als Client kommuniziert über

die Standardmethoden GET und POST über HTTP mit dem Server, wobei der Server seine Operationen über URIs (hier immer URLs) zur Verfügung stellt und Ressourcen als HTML-Repräsentationen dargestellt werden (vgl. [Starke 2018]).

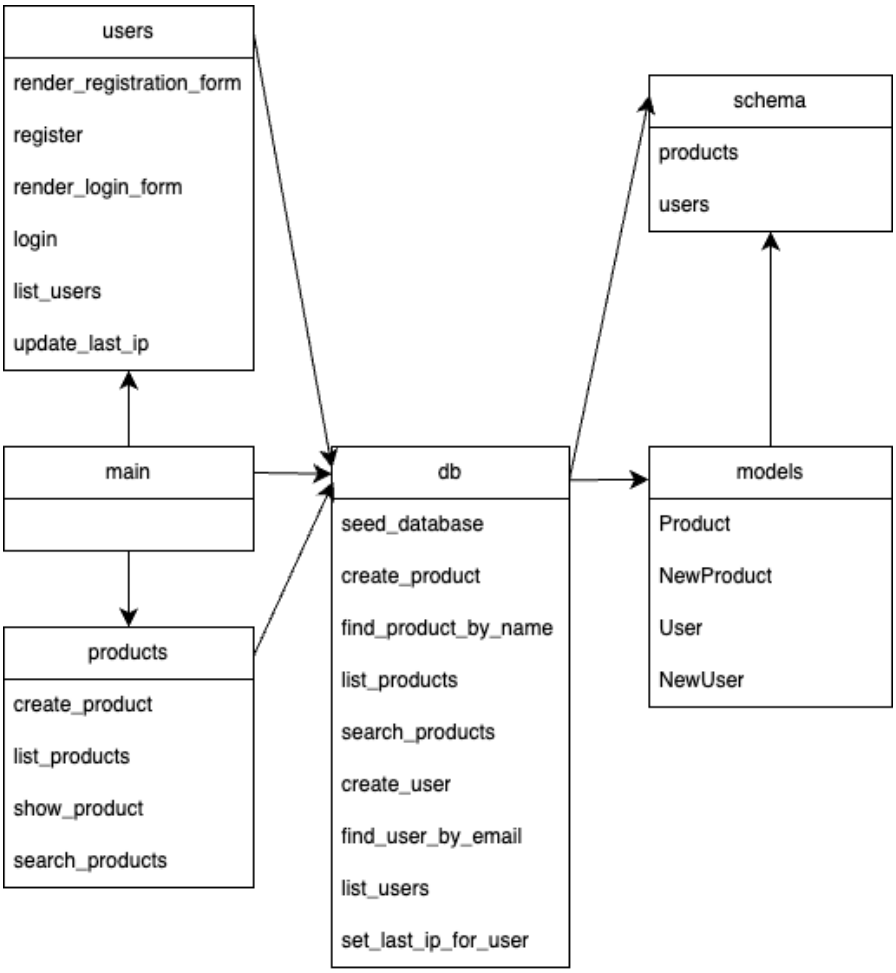

Abbildung 6.2: Bausteinübersicht Prototyp, Grafiktemplate: draw.io (CC BY 4.0)

Die *main*-Datei enthält die Routen zu den Services, die von der Anwendung angeboten werden. Sie hängt dabei von den Dateien *users* und *products* ab, die Funktionen für diese beiden Elemente der Domänenlogik bereitstellen. Sie hängt außerdem für den Programmstart von *db* ab, um die Datenbank zu befüllen.

Sowohl *users* als auch *products* verwenden *db*, um mit der Datenbank zu interagieren. Sie stellen die API-Endpunkte zur Verfügung. Die zentralen Aspekte der Fachdomäne – Produkt und Kund:in – werden in Rust als Structs in der Datei *models* abgebildet: *Product*

und *User*. *User* ist dabei nicht genau deckungsgleich mit *Kund:in*, da es auch andere Nutzer:innen (z. B. Admins) umfassen könnte. Rollen sind hier noch nicht umgesetzt. Da die ID erst beim Anlegen erstellt wird, ist jeweils ein weiteres Struct für ein neues Produkt bzw. eine:n neue:n Nutzer:in notwendig, wie in Abbildung 6.3 beispielhaft abgebildet. In Abbildung 6.3 wird auch die Verwendung von Diesel für die Anbindung der Datenbank deutlich. Der ORM Diesel generiert die Datei *schema*, die das Datenbankschema mit Hilfe von Macros in Rust darstellt.

```rust
#[derive(Queryable, Selectable, Serialize)]
#[diesel(table_name = products)]
#[diesel(check_for_backend(diesel::sqlite::Sqlite))]
pub struct Product {
    pub id: i32,
    pub name: String,
    pub description: String,
    pub price: i32,
    pub quantity: i32,
}

2 usages  Ann
#[derive(Insertable)]
#[diesel(table_name = products)]
pub struct NewProduct {
    pub name: String,
    pub description: String,
    pub price: i32,
    pub quantity: i32,
}
```

Abbildung 6.3: Screenshot: Prototyp, 2024-09-02

6.3 Implementierung

Als Backend-Framework wird das in Unterabschnitt 4.3.1 vorgestellte *actix-web* verwendet. Die Entscheidung für dieses Framework basiert auf seiner relativen Reife innerhalb des Rust-Ökosystems und seiner guten Dokumentation. Diese Einschätzung bestätigte sich im Laufe der Entwicklung des Prototyps, die durch die ausführliche und aktuelle Dokumentation erleichtert wurde.

Für die Persistierung von Daten wird SQLite eingesetzt, das eine relationale Datenbank zur Verfügung stellt, die SQL verwendet. SQLite ist laut der Firma Hwaci, die es entwickelt, die weltweit am häufigsten genutzte Datenbank-Engine [Hipp, Wyrick & Company, Inc. (Hwaci)].

Für die Kommunikation mit der Datenbank wird Diesel verwendet, ein in Rust geschriebener Object Relational Mapper und Query Builder, der als Open-Source-Projekt entwickelt wird [Diesel Core Team b]. Dabei wurde basierend auf dem *Getting Started with Diesel*-Handbuch des Diesel Core Teams [Diesel Core Team c], den Beispielen aus dem Github-Repository für Diesel [Diesel Contributors b] und der Diesel-Dokumentation [Diesel Core Team a] gearbeitet.

Für das Templating der angezeigten Seiten wird die Rust-Implementation von *Handlebars* verwendet, das HTML generiert [Katz]. Dabei bietet es bereits HTML-Bereinigung, die teilweise umgangen werden muss, wenn das Verhalten des darunterliegenden Rust-Programms beobachtet werden soll. Die Einbindung im Prototyp basiert auf den Beispielen zur Verwendung von Handlebars aus dem Github-Repository für actix-web [Actix Web Contributors]. HTML und CSS sind abgewandelt von Beispielen der *W3 Schools How To* [Refsnes Data] sowie der *MDN Web Docs* der Mozilla Corporation [Mozilla Corporation b].

Für die Registrierung von Nutzer:innen werden Benutzer:innenname, E-Mail-Adresse und Passwort verwendet und das für die *API-only-XSS*-Challenge notwendige Session Token durch ein Cookie umgesetzt.

Als Entwicklungsumgebung wurde die IDE *RustRover* von JetBrains genutzt.

7 Validierung und Vergleich

Der Prototyp ließ sich wie geplant umsetzen und erfüllt alle in Tabelle 6.3 zusammengefassten Anforderungen. In Tabelle 7.1 wird ausgeführt, wie die Anforderungen im Prototyp konkret umgesetzt wurden.

ID	Anforderung	Umsetzung
01	Webanwendung in Rust	Der Prototyp wurde bis auf Teile der Präsentationsschicht in Rust entwickelt.
02	Framework in Rust	siehe Tabelle 7.2
03	Bibliotheken in Rust	siehe Tabelle 7.2: Alle zentralen Tools sind zu mindestens 99 Prozent in Rust geschrieben.
04	Shop	Die Webanwendung zeigt eine Reihe von Produkten an und stellt dadurch einen Shop dar.
05	Nutzer:innen-Input	Nutzer:innen können in Formularen Text eingeben.
06	Speicherung und Abruf von Nutzer:inneneingaben	Durch Formulare sowie einen API-Endpunkt übermittelte Eingaben werden in einer Datenbank gespeichert.
07	Registrierung und Login	Nutzer:innen können sich in entsprechenden Formularen registrieren. Die Daten werden in einer Datenbank gespeichert und Nutzer:innen können sich danach damit einloggen.
08	clientseitige Validierung von Eingaben	Ein Feld, in dem eine E-Mail-Adresse eingegeben werden soll, wird clientseitig darauf überprüft.
09	eigener HTTP-Header für Client-IP-Adresse	Der Server liest einen eigenen Header und speichert die Information, ohne sie zu überprüfen.
10	manipulierbare URL	Die URL für ein Produkt enthält dessen Namen.
11	Suchanfragen in URL	Die Suche nach Produkten wird clientseitig durchgeführt und die Anfrage so in die URL eingefügt.

Tabelle 7.1: Umsetzung der Anforderungen

Tool	Verwendungszweck	Rust-Anteil in % (GitHub-Repository, Stand 2024-09-22)
actix-web	Web-Framework	99,4 % [Actix Contributors]
Diesel	Object Relational Mapper	99,9 % [Diesel Contributors a]
Serde	Umwandlung in JSON / aus JSON	100 % [Serde Contributors]
bcrypt-rust	Hashing und Überprüfung von Passwörtern	100 % [Rust-bcrypt Contributors]
handlebars-rust	HTML-Templates	99 % [Handlebars-rust Contributors]

Tabelle 7.2: Verwendete Tools

Auch die Angriffe, die in den Challenges durchgeführt wurden, waren erfolgreich. Es wurde hiermit deutlich, dass auch eine in Rust geschriebene Webanwendung durch Cross-Site Scripting kompromittiert werden kann. Im Folgenden wird genauer auf die aus dem Juice Shop in den Prototyp übertragenen Angriffe eingegangen.

7.1 Umsetzung und Erfolg der Angriffe

Kategorie	Challenge	Angriff erfolgreich
Stored XSS	Stored XSS[a]	ja
Stored XSS	API-only XSS	ja
Stored XSS	Client-side XSS Protection	ja
Stored XSS	HTTP-Header XSS	ja
Reflected XSS	Reflected XSS	ja
DOM-based XSS	DOM XSS	ja

[a] eigenes Minimalbeispiel

Tabelle 7.3: Erfolg der Angriffe

Wie in Tabelle 7.3 dargestellt ist und im Folgenden genauer beschrieben wird, waren alle aus dem Juice Shop übernommenen XSS-Angriffe sowie der selbst daraus abgelei-

tete Angriff erfolgreich. Als Script-Befehl wurde in allen Angriffen der im Juice Shop vorgeschlagene Befehl `<iframesrc="javascript:alert('xss')">` verwendet.

7.1.1 Stored XSS

Stored XSS

Um zunächst festzustellen, ob ein einfacher Stored-XSS-Angriff erfolgreich ist, wurde aus den Angriffen dieser Juice-Shop-Kategorie, die alle jeweils zusätzliche Aspekte beeinhalten, ein Minimalbeispiel abgeleitet. Hierfür wird im Namensfeld des Registrierungsformulars der Script-Befehl eingegeben. Bei Aufruf der Nutzer:innenliste sollte es zur Ausführung kommen. Zunächst war dieser Angriff nicht erfolgreich – die Eingaben wurden bereinigt und, wie in Abbildung 7.1 dargestellt, nur als Text angezeigt. Der Grund dafür liegt in der Verwendung von Handlebars für das HTML Templating: Handlebars bietet *HTML Escaping*, wandelt also Sonderzeichen in ihre HTML-Versionen um [Katz].

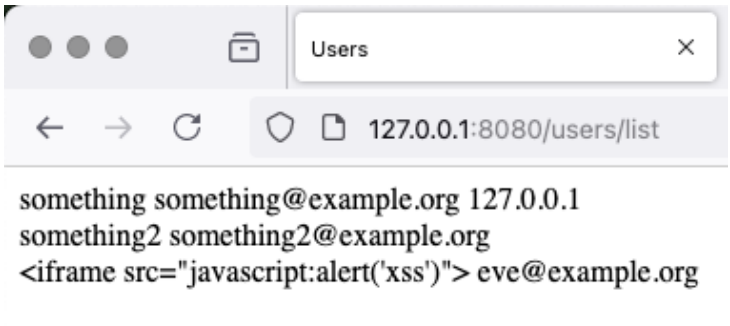

Abbildung 7.1: Screenshot: Prototyp mit HTML-Escaping, 2024-09-07

Um das Verhalten des Rust-Programms beobachten zu können und die Anwendung, wenn möglich, dem Juice Shop ähnlich angreifbar zu machen, wurde in Handlebars die ebenfalls vorgesehene Darstellung von Eingaben ohne *HTML Escaping* verwendet – woraufhin der Angriff gelang, wie in Abbildung 7.2 abgebildet.

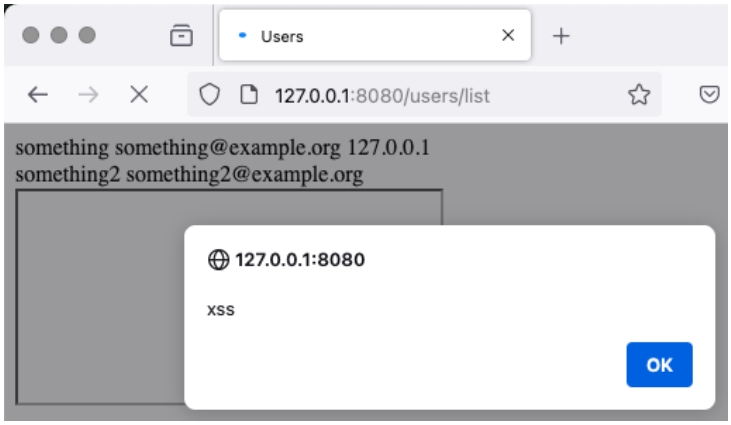

Abbildung 7.2: Screenshot: Prototyp ohne HTML-Escaping, 2024-09-07

API-only XSS

Wie im Juice Shop wird ein API-Endpunkt zur Verfügung gestellt, der nicht über die grafische Oberfläche zu erreichen ist. Dies ermöglicht es, ein Produkt hinzuzufügen. Mit Hilfe der Browserkonsole oder eines geeigneten Programms wie etwa cURL kann somit ein Produkt hinzugefügt werden, das in der Beschreibung einen Script-Befehl enthält (Abbildung 7.3). Dabei muss in cURL, ähnlich wie in der Juice-Shop-Challenge, der Session-Cookie aus einem vorigen Request mitgesendet werden, da nur eingeloggte Nutzer:innen die Aktion durchführen dürfen. Verwendet man `fetch` in der Browserkonsole, wie in Abbildung 7.3 gezeigt, geschieht dies von selbst.

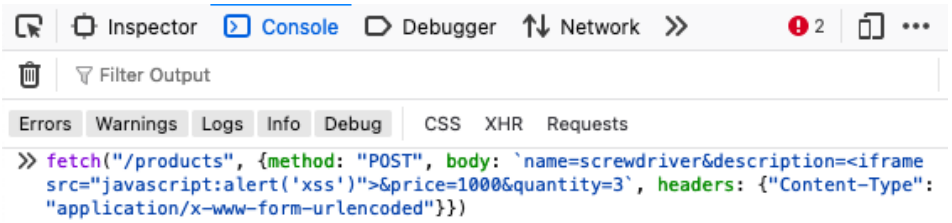

Abbildung 7.3: Screenshot: Prototyp (Request), 2024-09-07

Wird die Liste der Produkte aufgerufen, wird deutlich, dass der Angriff erfolgreich war: Das Script wird ausgeführt, wie in Abbildung 7.4 ersichtlich ist.

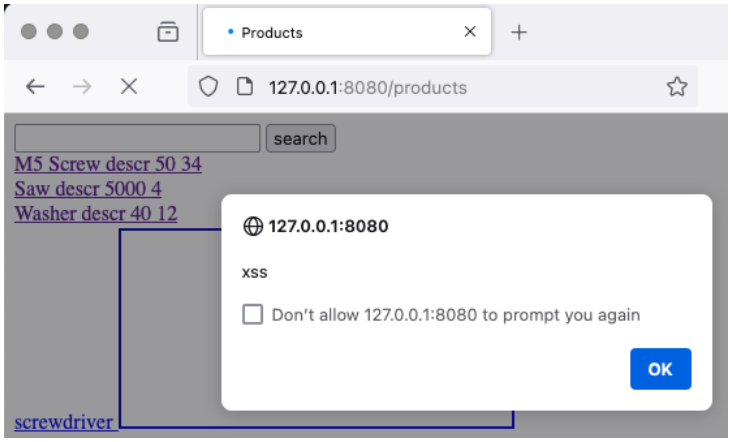

Abbildung 7.4: Screenshot: Prototyp (API-only XSS), 2024-09-07

Client-side XSS Protection

Bei der Registrierung wird durch die Verwendung des HTML-Input-Typs *email* erzwungen, dass Nutzer:innen eine der festgelegten Form entsprechende E-Mail-Adresse angeben. Dies lässt sich umgehen, indem der Request nicht über die Eingabemaske, sondern direkt gesendet wird – z. B. wie in Abbildung 7.5 mit Hilfe der Browserkonsole.

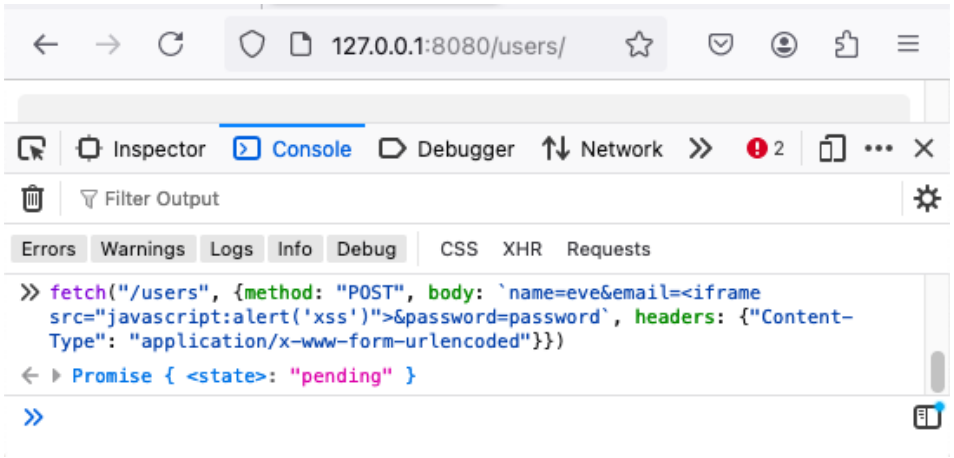

Abbildung 7.5: Screenshot: Prototyp (Request), 2024-09-07

Dadurch wird die clientseitige Eingabevalidierung umgangen und der Angriff ist erfolgreich. Da hier die Validierung durch HTML stattfindet, hat die Verwendung von Rust keinen Einfluss auf den Angriff.

Abbildung 7.6: Screenshot: Prototyp (Client-side XSS Protection), 2024-09-07

HTTP-Header XSS

Um die Schwachstelle aus der Juice-Shop-Challenge nachzubilden, wurde ein eigener Header mit dem Namen *"True-Client-IP"* verwendet. Der Server liest diesen und speichert die dort angegebene Information, prüft aber – wie im Vorbild – nicht, ob tatsächlich ein entsprechendes Format vorliegt. Dadurch kann ein Request, wie in Abbildung 7.8 abgebildet, direkt an den API-Endpunkt gesendet werden, der anstelle einer IP-Adresse einen Script-Befehl enthält.

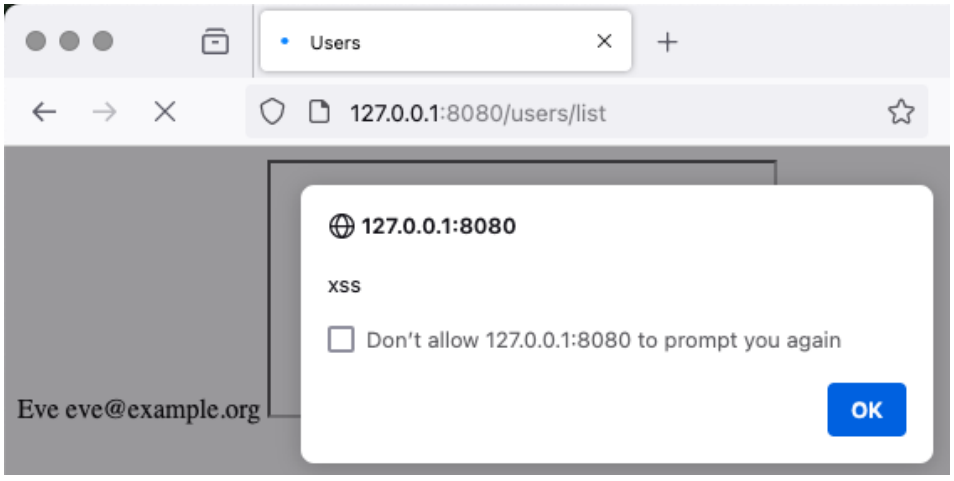

Abbildung 7.7: Screenshot: Prototyp (HTTP-Header XSS – Request), 2024-09-07

Wird danach die Liste der Nutzer:innen aufgerufen, kommt das Script zur Ausführung, wie in Abbildung 7.7 dargestellt. Hier ist auch ersichtlich, dass Firefox über einen Me-

chanismus verfügt, um zu erkennen, wenn eine Webanwendung zahlreiche Benachrichtigungsdialoge öffnet, und Nutzer:innen erlaubt, diese zu unterdrücken.

Abbildung 7.8: Screenshot: Prototyp (HTTP-Header XSS), 2024-09-07

7.1.2 Reflected XSS

Anstatt dem Juice-Shop-Beispiel genau zu folgen und eine Paketverfolgung mit Sendungsnummer in der URL umzusetzen, für die ein kompletter Bestellprozess mit Adresse und Zahlungsart notwendig wäre, wurden für diesen Angriff die Produktdetailseiten verwendet. Werden Produkte über ihre ID aufgerufen, ist der Angriff nicht möglich, da hier der Typ der ID (`i32`) nicht durch das Script erfüllt wird. Hier wird deutlich, dass das Typsystem durchaus helfen kann, unerwünschte Eingaben zu verhindern. Bei Verwendung des Namens für den Zugriff auf ein Produkt – in einem echten Shop unpraktisch, da sich Namen ändern können, ohne dass sich das eigentliche Produkt ändert – kann der Script-Befehl anstelle des Namens, der Bestandteil der URL ist, eingefügt werden.

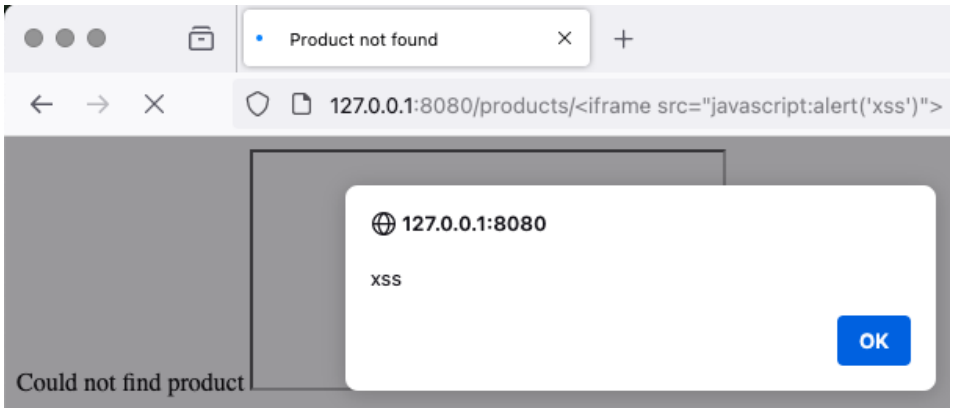

Abbildung 7.9: Screenshot: Prototyp (Reflected XSS), 2024-09-07

7.1.3 DOM-based XSS

Wie in Unterabschnitt 3.2.3 beschrieben, wird die Suche nach Produkten clientseitig durchgeführt, statt eine Anfrage an den Server zu senden. So kann ein Script-Befehl als Suchbegriff verwendet werden, der dadurch in das DOM eingefügt wird – der Angriff gelingt, wie in Abbildung 7.10 dargestellt. Hier war zu erwarten, dass es keine Unterschiede zwischen dem Juice Shop und dem Prototyp gibt, da der Angriff komplett auf der Ebene des DOM stattfindet.

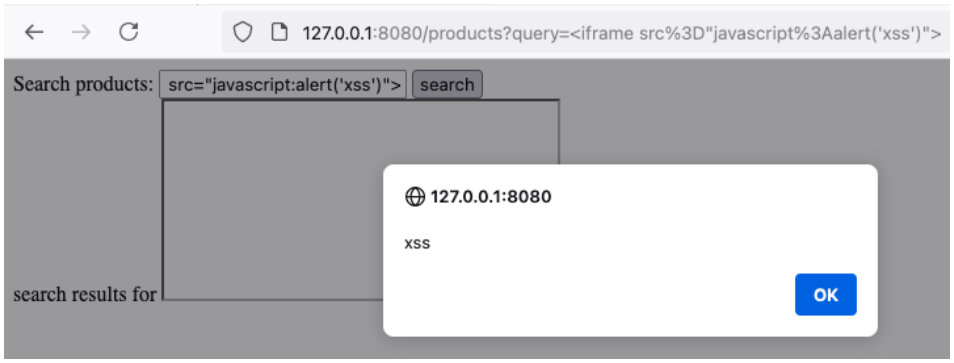

Abbildung 7.10: Screenshot: Prototyp (DOM XSS), 2024-09-07

7.2 Sicherheitstools

Da alle Angriffe erfolgreich waren, müssen als Schutz vor Cross-Site Scripting zusätzliche Werkzeuge eingesetzt werden. Aus der Kategorie der allgemeinen Sicherheitswerkzeuge für Rust (Unterabschnitt 4.4.1) sind hierfür die Bibliotheken *sanitizer* und *validator* interessant, die jedoch mit eigenen Regeln für Validierung und Bereinigung verwendet werden müssten, wenn sie für Textfelder und nicht für Standardtypen wie etwa E-Mail-Adressen eingesetzt werden sollen. *Mirchecker*, *Miri* und *cargo-careful* zielen auf undefiniertes Verhalten auf Rust-Ebene, das für XSS nicht relevant ist, und können hier daher nicht weiterhelfen.

Unter den XSS-spezifischen Werkzeugen (Unterabschnitt 4.4.2) sind *sanitize_html* und *ammonia* interessant. Der Einsatz einer Content Security Policy (CSP) ist zwar ebenfalls sinnvoll, der entsprechende Header kann aber mit dem verwendeten Web-Framework gesetzt werden. Dies macht den Einsatz von *http-types* unnötig, da es lediglich Komplexität hinzufügen würde. *Libinjection-rs* kann bekannte XSS-Angriffe erkennen, könnte also zur

Validierung etwa beim Anlegen eines neuen Nutzer:innenaccounts eingesetzt werden. Es bietet jedoch nur einen geringen Schutz, wie in Unterabschnitt 4.4.2 ausgeführt.

Dem Produkt-Endpoint wurde ein Query-Parameter hinzugefügt, der das Ausprobieren von *ammonia* und *sanitize_html* möglich macht. In beiden Fällen wird die Eingabe erfolgreich bereinigt und der Angriff schlägt fehl, wie in Abbildung 7.11 und Abbildung 7.12 zu sehen ist.

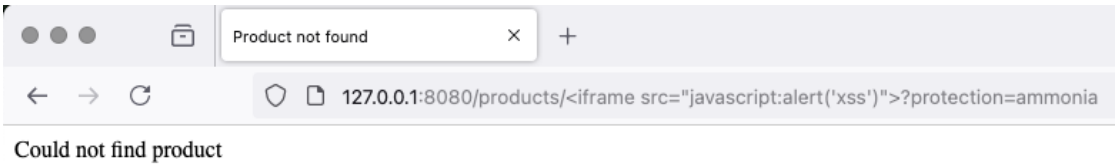

Abbildung 7.11: Screenshot: Prototyp mit sanitization (ammonia), 2024-09-07

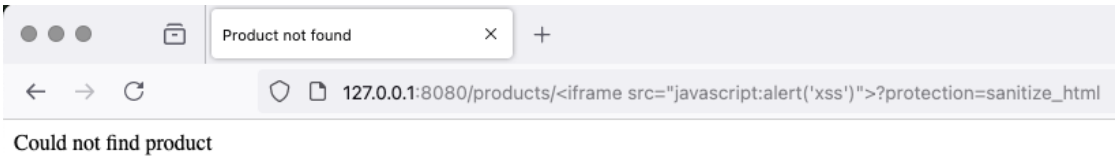

Abbildung 7.12: Screenshot: Prototyp mit sanitization (sanitize_html), 2024-09-07

7.3 Erkenntnisse aus dem Prototyp

Alle Juice-Shop-Challenges, die im Hinblick auf ihre Funktionsweise umsetzbar waren, konnten erfolgreich in den Prototyp einer angreifbaren Webanwendung integriert werden. Weder Rust noch das verwendete Framework verhindern XSS-Angriffe. Lediglich das verwendete Templating hatte hier Erfolg: Es verhinderte in der Grundeinstellung den Erfolg von Angriffen, da es die Eigaben bereinigte. Dies ist für Reflected XSS und Stored XSS relevant, jedoch ein sprachunabhängiger Mechanismus. Um die Anwendung dem Vorbild gemäß angreifbar zu machen, konnte dies einfach umgangen werden. Dabei wurde deutlich, dass der Unterschied in der Handlebars-Schreibweise gering und unauffällig ist: Es handelt sich nur um die Anzahl der geschweiften Klammern, wodurch die Gefahr nicht, wie etwa bei `dangerouslySetInnerHTML` in React, explizit gemacht wird [Meta Platforms Inc.]. Besteht eine erwartete Eingabe nicht aus Text, kann Rusts Typsystem

verhindern, dass stattdessen ein Script gespeichert wird. Obwohl im Juice Shop mit dem *Angular*-Framework das ebenfalls statisch typisierte *TypeScript* verwendet wird (vgl. Abschnitt 5.2), wäre dies dort nicht der Fall, da TypeScript bei der Kompilierung in JavaScript umgewandelt wird, das keine Typgarantien gibt [Microsoft Corporation]. Da in den meisten Fällen Nutzer:inneneingaben aus Text bestehen, ist dieses Verhalten von Rust für den Schutz vor XSS jedoch im Normalfall nicht hilfreich. Rust hat also keine inhärenten Vorteile im Hinblick auf diese Art von Angriffen.

8 Fazit und Ausblick

Wie im Prototyp deutlich wurde, kann Rust beim Schutz von Webanwendungen gegen Angriffe durch Cross-Site Scripting kaum mehr Unterstützung leisten als andere Sprachen. Der einzige Vorteil, der bei den durchgeführten Angriffen zum Tragen kam, besteht darin, dass Rusts Typsystem davor schützt, statt einer als Integer definierten ID einen Script-Befehl zu verarbeiten. Da die meisten Eingaben, die in einer Webanwendung zu erwarten sind, jedoch aus Text bestehen, ist dies nur sehr eingeschränkt hilfreich. Wo immer solche Eingaben verarbeitet und verwendet werden, ist also besondere Vorsicht und der Einsatz spezieller Werkzeuge wie der vorgestellten und exemplarisch angewandten Rust-Bibliotheken zum Schutz vor XSS notwendig. Cross-Site Scripting gehört damit zu den Schwachstellen, bei denen [Sible und Svoboda 2022] Rust-Entwickler:innen dazu mahnen, wachsam zu bleiben. Ähnliches gilt vermutlich auch für die weiteren Angriffe aus der OWASP-Top-Ten-Kategorie *Injection*. Es ist zu vermuten, dass Rust in den anderen Kategorien ebenfalls über keine Vorteile anderen Sprachen gegenüber verfügt, da diesen generell sprachunabhängige Programmier- und Designfehler zugrunde liegen. Denkbar ist, dass Rust dabei helfen könnte, Schwachstellen der Kategorie *Vulnerable and Outdated Components* zu verhindern, indem einzelne Komponenten von Anwendungen durch die in Abschnitt 4.1 vorgestellten Prinzipien vor auszunutzenden Fehlern geschützt werden.

Wie aufgezeigt, ist die Umsetzung einer angreifbaren Webanwendung in Rust möglich. Der Prototyp wäre analog zum Juice Shop um weitere Kategorien erweiterbar. Da sich jedoch herausstellte, dass der Einsatz von Rust hier wenig Unterschied machte, hängt die Wahl der Sprache für mögliche andere Webanwendungen nicht von ihrer Sicherheit, sondern ihren anderen Merkmalen ab. In der Programmierung erwies sich Rust als gut nutzbar und auch der „Kampf mit dem *Borrow Checker*“ eher als nützlich denn als störend. Lediglich der Umgang mit Strings ist unkomfortabler als etwa in Java, da der Einsatz von `String` oder `&str`, teilweise mit notwendigen Angaben zur *lifetime*, komplexer ist. Hier ist die in Abschnitt 4.1 angesprochene Verlagerung der „Schmerzen“, die ein Programm verursacht, in die Phase der Entwicklung spürbar.

Das gewählte Web-Framework *actix-web* verfügt über alle im Prototyp notwendigen Funktionen und eine außergewöhnlich gute Dokumentation. Es ist im hier verwendeten Rahmen in der Einfachheit der Anwendung durchaus mit dem verbreiteten Java-Framework Spring Boot vergleichbar. Rust eignet sich also – auch wenn es hier keine besonderen Vorteile im Schutz vor Cross-Site Scripting zeigen konnte – von seiner Programmierung für die Entwicklung von Webanwendungen und hat aufgrund seines Fokus auf Nebenläufigkeit gerade für die Untergruppe der Webanwendungen, für die Performanz wichtig ist, Potenzial.

Literaturverzeichnis

[Actix Contributors] Actix Contributors: *Actix Web*. – URL https://github.com/actix/actix-web. – Zugriffsdatum: 2024-07-13

[Actix Team] Actix Team: *Actix Web Docs*. – URL https://actix.rs/docs. – Zugriffsdatum: 2024-07-13

[Actix Web Contributors] Actix Web Contributors: *Handlebars*. – URL https://github.com/actix/examples/tree/master/templating/handlebars. – Zugriffsdatum: 2024-09-01

[Alcorn u. a. 2014] Alcorn, W. ; Frichot, C. ; Orru, M.: *The Browser Hacker's Handbook*. Wiley, 2014

[Amberg und Schmid 2024] Amberg, E. ; Schmid, D.: *Hacking: Der umfassende Praxis-Guide*. MITP Verlags GmbH, 2024 (mitp Professional)

[Ammonia Contributors] Ammonia Contributors: *Ammonia. HTML Sanitization.* – URL https://github.com/rust-ammonia/ammonia. – Zugriffsdatum: 2024-07-17

[Andress 2014] Andress, J.: *The Basics of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice*. Elsevier Science, 2014

[Armor Contributors 2020] Armor Contributors: *Armor*. 2020. – URL https://github.com/http-rs/armor. – Zugriffsdatum: 2024-07-22

[Axum Contributors] Axum Contributors: *Axum*. – URL https://github.com/tokio-rs/axum. – Zugriffsdatum: 2024-07-13

[Baumgartner 2023] Baumgartner, S.: *Best Rust Web Frameworks to Use in 2023*. 2023. – URL https://www.shuttle.rs/blog/2023/08/23/rust-web-framework-comparison. – Zugriffsdatum: 2024-07-13

[Baun 2020] Baun, C. ; Wiesbaden, Springer F. (Hrsg.): *Operating Systems / Betriebssysteme Bilingual Edition: English – German / Zweisprachige Ausgabe: Englisch – Deutsch*. Springer Vieweg, 2020 (Lehrbuch). – URL https://doi.org/10.1007/978-3-658-29785-5

[Blandy u. a. 2021] Blandy, J. ; Orendorff, J. ; Tindall, L.: *Programming Rust*. 2. Auflage. 2021

[Cargo-careful Contributors a] Cargo-careful Contributors: *Cargo-careful*. – URL https://crates.io/crates/cargo-careful. – Zugriffsdatum: 2024-07-22

[Cargo-careful Contributors b] Cargo-careful Contributors: *Cargo-careful*. – URL https://github.com/RalfJung/cargo-careful. – Zugriffsdatum: 2024-07-22

[Cargo Contributors] Cargo Contributors: *The Cargo Book*. – URL https://doc.rust-lang.org/cargo/reference/manifest.html. – Zugriffsdatum: 2024-07-13

[Crichton 2014] Crichton, A.: *Cargo: Rust's community crate host*. 2014. – URL https://blog.rust-lang.org/2014/11/20/Cargo.html. – Zugriffsdatum: 2024-06-01

[Diesel Contributors a] Diesel Contributors: *Diesel*. – URL https://github.com/diesel-rs/diesel. – Zugriffsdatum: 2024-09-22

[Diesel Contributors b] Diesel Contributors: *Diesel/examples*. – URL https://github.com/diesel-rs/diesel/tree/2.1.x/examples. – Zugriffsdatum: 2024-07-24

[Diesel Core Team a] Diesel Core Team: *API Documentation*. – URL https://docs.diesel.rs. – Zugriffsdatum: 2024-07-24

[Diesel Core Team b] Diesel Core Team: *Diesel*. – URL https://diesel.rs. – Zugriffsdatum: 2024-07-24

[Diesel Core Team c] Diesel Core Team: *Getting Started with Diesel*. – URL https://diesel.rs/guides/getting-started. – Zugriffsdatum: 2024-07-24

[Dioxus Contributors] Dioxus Contributors: *Dioxus*. – URL https://github.com/dioxuslabs/dioxus. – Zugriffsdatum: 2024-07-13

[Eckert 2023] Eckert, C.: *IT-Sicherheit: Konzepte – Verfahren – Protokolle*. De Gruyter Oldenbourg, 2023 (De Gruyter Studium)

[Flitton 2023] Flitton, M.: *Rust Web Programming*. 2. Auflage. Packt Publishing Limited, 2023

[Gebeshuber u. a. 2019] Gebeshuber, K. ; Teiniker, E. ; Zugaj, W.: *Exploit! Code härten, Bugs analysieren, Hacks verstehen*. Rheinwerk Verlag, 2019 (Rheinwerk Computing)

[Grassi u. a. 2017] Grassi, P. ; Fenton, J. ; Garcia, M.: *Digital Identity Guidelines*. 2017-12-01 2017

[Handlebars-rust Contributors] Handlebars-rust Contributors: *Handlebars-rust*. – URL https://github.com/sunng87/handlebars-rust. – Zugriffsdatum: 2024-09-22

[Hipp, Wyrick & Company, Inc. (Hwaci)] Hipp, Wyrick & Company, Inc. (Hwaci): *SQLite*. – URL https://www.sqlite.org. – Zugriffsdatum: 2024-09-01

[Hoffman 2024] Hoffman, A.: *Web Application Security*. O'Reilly Media, 2024

[Http-Types Contributors] Http-Types Contributors: *Http-Types*. – URL https://github.com/http-rs/http-types. – Zugriffsdatum: 2024-07-22

[IEEE 1990] IEEE: IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std 610.12-1990* (1990), S. 1–84

[IEEE 2017] IEEE: ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. In: *ISO/IEC/IEEE 24765:2017(E)* (2017), S. 1–541

[Jones und Hardt 2012] Jones, M. ; Hardt, D.: *IETF RFC 6750 The OAuth 2.0 Authorization Framework: Bearer Token Usage*. 2012. – URL https://datatracker.ietf.org/doc/html/rfc6750. – Zugriffsdatum: 2024-07-24

[Juice Shop Contributors] Juice Shop Contributors: *github.com/juice-shop/juice-shop*. – URL https://github.com/juice-shop/juice-shop. – Zugriffsdatum: 2024-06-18

[Jung 2020] Jung, R.: *Understanding and evolving the Rust programming language*. Saarbrücken, Universität des Saarlandes, PhD thesis, August 2020. – https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

[Katz] Katz, Y.: *Handlebars. Minimal Templating on Steroids.* – URL https://handlebarsjs.com. – Zugriffsdatum: 2024-09-01

[Kimminich] Kimminich, B.: *Pwning OWASP Juice Shop.* – URL https://pwning.owasp-juice.shop. – Zugriffsdatum: 2024-06-29

[Klabnik und Nichols] Klabnik, S. ; Nichols, C.: *The Rust Programming Language.* – URL https://github.com/rust-lang/book/tree/main/first-edition. – Zugriffsdatum: 2024-07-13

[Klabnik und Nichols 2023] Klabnik, S. ; Nichols, C.: *The Rust Programming Language.* 2. Auflage. No Starch Press, 2023

[Kofler u. a. 2023] Kofler, M. ; Gebeshuber, K. ; Aigner, R. ; Kloep, P. ; Hackner, T. ; Neugebauer, F. ; Kania, S. ; Scheible, T. ; Zingsheim, A. ; Widl, M.: *Hacking & Security: das umfassende Handbuch.* Rheinwerk Verlag, 2023 (Rheinwerk Computing)

[Leptos Contributors] Leptos Contributors: *Leptos.* – URL https://github.com/leptos-rs/leptos. – Zugriffsdatum: 2024-07-13

[Lerche] Lerche, C.: *Tokio Tutorial.* – URL https://tokio.rs/tokio/tutorial. – Zugriffsdatum: 2024-07-13

[Li u. a. 2021] Li, Z. ; Wang, J. ; Sun, M. ; Lui, J.: MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* New York, NY, USA : Association for Computing Machinery, 2021 (CCS '21), S. 2183–2196. – URL https://doi.org/10.1145/3460120.3484541

[LibInjection Contributors] LibInjection Contributors: *<'/> | libInjection.* – URL https://github.com/libinjection/libinjection. – Zugriffsdatum: 2024-07-24

[LibInjection-Rs Contributors] LibInjection-Rs Contributors: *Libinjection-rs.* – URL https://github.com/arvancloud/libinjection-rs. – Zugriffsdatum: 2024-07-24

[Lyu 2021] Lyu, S.: *Practical Rust Web Projects: Building Cloud and Web-Based Applications.* Apress, 2021

[Matsakis 2016] Matsakis, N.: *Introducing MIR*. 2016. – URL https://blog.rust-lang.org/2016/04/19/MIR.html. – Zugriffsdatum: 2024-07-17

[Matsakis 2017] Matsakis, N.: *Rust: Hack Without Fear!* Konferenzvortrag, C++Now 2017, Denver. 06 2017. – URL https://www.youtube.com/watch?v=l01z-7cuRYI. – Zugriffsdatum: 2024-07-13

[McDonald 2020] McDonald, M.: *Web Security for Developers: Real Threats, Practical Defense*. No Starch Press, 2020

[Meta Platforms Inc.] Meta Platforms Inc.: *React API Reference. Common components (e.g.): Dangerously setting the inner HTML*. – URL https://react.dev/reference/react-dom/components/common#dangerously-setting-the-inner-html. – Zugriffsdatum: 2024-09-23

[Microsoft Corporation] Microsoft Corporation: *Type Script*. – URL https://www.typescriptlang.org. – Zugriffsdatum: 2024-09-23

[Miri Contributors] Miri Contributors: *Miri*. – URL https://github.com/rust-lang/miri. – Zugriffsdatum: 2024-07-22

[Mozilla Corporation a] Mozilla Corporation: *Content Security Policy (CSP)*. – URL https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. – Zugriffsdatum: 2024-05-06

[Mozilla Corporation b] Mozilla Corporation: *MDN Web Docs*. – URL https://developer.mozilla.org/en-US/docs/Web. – Zugriffsdatum: 2024-09-01

[Mozilla Corporation c] Mozilla Corporation: *MDN Web Docs: WebAssembly Concepts*. – URL https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. – Zugriffsdatum: 2024-09-01

[Mozilla Corporation d] Mozilla Corporation: *webassambly.org*. – URL https://webassembly.org/. – Zugriffsdatum: 2024-07-13

[Mozilla Corporation e] Mozilla Corporation: *X-XSS-Protection*. – URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection. – Zugriffsdatum: 2024-07-22

[Najera-Gutierrez 2018] Najera-Gutierrez, G.: *Kali Linux Web Penetration Testing Cookbook: Identify, exploit, and prevent web application vulnerabilities with Kali Linux 2018.x*. 2018

[OWASP Foundation a] OWASP Foundation: *Cross Site Scripting Prevention Cheat Sheet*. – URL https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html. – Zugriffsdatum: 2024-08-23

[OWASP Foundation b] OWASP Foundation: *Cross Site Scripting (XSS)*. – URL https://owasp.org/www-community/attacks/xss/. – Zugriffsdatum: 2024-05-06

[OWASP Foundation c] OWASP Foundation: *DOM Based XSS*. – URL https://owasp.org/www-community/attacks/DOM_Based_XSS. – Zugriffsdatum: 2024-05-06

[OWASP Foundation d] OWASP Foundation: *Projects*. – URL https://owasp.org/projects/. – Zugriffsdatum: 2024-06-28

[OWASP Foundation e] OWASP Foundation: *Types of XSS*. – URL https://owasp.org/www-community/Types_of_Cross-Site_Scripting. – Zugriffsdatum: 2024-05-06

[OWASP Foundation 2017] OWASP Foundation: *OWASP TopTen*. 2017. – URL https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS).html. – Zugriffsdatum: 2024-05-06

[OWASP Foundation 2021a] OWASP Foundation: *OWASP Top 10 Privacy Risks*. 2021. – URL hhttps://owasp.org/www-project-top-10-privacy-risks/. – Zugriffsdatum: 2024-06-18

[OWASP Foundation 2021b] OWASP Foundation: *OWASP TopTen*. 2021. – URL https://owasp.org/www-project-top-ten/. – Zugriffsdatum: 2024-05-06

[OWASP Foundation 2021c] OWASP Foundation: *OWASP TopTen*. 2021. – URL https://owasp.org/Top10/A03_2021-Injection/. – Zugriffsdatum: 2024-05-06

[OWASP Foundation 2023] OWASP Foundation: *OWASP API Security Project*. 2023. – URL https://owasp.org/www-project-automated-threats-to-web-applications/. – Zugriffsdatum: 2024-06-18

[Pfleeger u. a. 2023] Pfleeger, C. ; Pfleeger, S. L. ; Coles-Kemp, L.: *Security in Computing*. Pearson Education, 2023

[Prasad 2016] Prasad, P.: *Mastering Modern Web Penetration Testing*. Packt Publishing, Limited, 2016

[Preston-Werner a] Preston-Werner, T.: *Semantic Versioning 2.0.0*. – URL https://semver.org. – Zugriffsdatum: 2024-06-05

[Preston-Werner b] Preston-Werner, T.: *TOML: Tom's Obvious Minimal Language*. – URL https://toml.io. – Zugriffsdatum: 2024-06-03

[Refsnes Data] Refsnes Data: *W3Schools How To*. – URL https://www.w3schools.com/howto. – Zugriffsdatum: 2024-09-01

[Rocket Contributors] Rocket Contributors: *Rocket*. – URL https://github.com/rwf2/Rocket. – Zugriffsdatum: 2024-07-13

[Rust-bcrypt Contributors] Rust-bcrypt Contributors: *Bcrypt*. – URL https://github.com/Keats/rust-bcrypt. – Zugriffsdatum: 2024-09-22

[Rust Contributors] Rust Contributors: *The Rust Programming Language*. – URL https://github.com/rust-lang/rust. – Zugriffsdatum: 2024-07-13

[Rust Core Team 2020] Rust Core Team: *Laying the foundation for Rust's future*. 2020. – URL https://blog.rust-lang.org/2020/08/18/laying-the-foundation-for-rusts-future.html. – Zugriffsdatum: 2024-06-01

[Rust Team] Rust Team: *Module std::sync*. – URL https://doc.rust-lang.org/stable/std/sync. – Zugriffsdatum: 2024-09-11

[Sanitize_html Contributors] Sanitize_html Contributors: *Crate sanitize_html*. – URL https://docs.rs/sanitize_html/latest/sanitize_html. – Zugriffsdatum: 2024-07-24

[Serde Contributors] Serde Contributors: *Serde*. – URL https://github.com/serde-rs/serde. – Zugriffsdatum: 2024-09-22

[Sharma u. a. 2019] Sharma, R. ; Kaihlavirta, V. ; Matzinger, C.: *The Complete Rust Programming Reference Guide: Design, develop, and deploy effective software systems using the advanced constructs of Rust*. Packt Publishing, 2019

[Sible und Svoboda 2022] Sible, J. ; Svoboda, D.: *Rust Software Security: A Current State Assessment*. Carnegie Mellon University, Software Engineering Institute's Insights (blog). Dec 2022. – URL https://doi.org/10.58012/0px4-9n81. – Zugriffsdatum: 2024-06-03

[Stack Overflow 2015] Stack Overflow: *Developer Survey Results*. 2015. – URL https://survey.stackoverflow.co/2015. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2016] Stack Overflow: *Developer Survey Results*. 2016. – URL https://survey.stackoverflow.co/2016. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2017] Stack Overflow: *Developer Survey Results*. 2017. – URL https://survey.stackoverflow.co/2017. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2018] Stack Overflow: *Developer Survey Results*. 2018. – URL https://survey.stackoverflow.co/2018. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2019] Stack Overflow: *Developer Survey Results*. 2019. – URL https://survey.stackoverflow.co/2018. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2020] Stack Overflow: *Developer Survey Results*. 2020. – URL https://survey.stackoverflow.co/2020. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2021] Stack Overflow: *Developer Survey Results*. 2021. – URL https://survey.stackoverflow.co/2021. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2022] Stack Overflow: *Developer Survey Results*. 2022. – URL https://survey.stackoverflow.co/2018. – Zugriffsdatum: 2024-07-13

[Stack Overflow 2023] Stack Overflow: *Developer Survey Results*. 2023. – URL https://survey.stackoverflow.co/2023/. – Zugriffsdatum: 2024-07-13

[Starke 2018] Starke, G.: *Effektive Softwarearchitekturen: ein praktischer Leitfaden*. Hanser, 2018

[Stouffer u. a. 2023] Stouffer, K. ; Pillitteri, V. ; Pease, M. ; Lightman, S. ; Tang, C. ; Zimmerman, T. ; Hahn, A. ; Saravia, S. ; Sherule, A. ; Thompson, M.: *Guide to Operational Technology (OT) Security*. 2023

[Stuttard und Pinto 2011] Stuttard, D. ; Pinto, M.: *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley, 2011

[Tech Empower 2023] Tech Empower: *Tech Empower Web Framework Benchmarks Round 22*. 2023. – URL https://www.techempower.com/benchmarks/#section=data-r22&test=composite&hw=ph. – Zugriffsdatum: 2024-07-13

[The MITRE Corporation a] The MITRE Corporation: *CWE-415: Double Free*. – URL https://cwe.mitre.org/data/definitions/415.html. – Zugriffsdatum: 2024-09-02

[The MITRE Corporation b] The MITRE Corporation: *CWE-416: Use After Free*. – URL https://cwe.mitre.org/data/definitions/416.html. – Zugriffsdatum: 2024-09-02

[The MITRE Corporation c] The MITRE Corporation: *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*. – URL https://cwe.mitre.org/data/definitions/79.html. – Zugriffsdatum: 2024-05-06

[The MITRE Corporation 2023] The MITRE Corporation: *2023 CWE Top 25 Most Dangerous Software Weaknesses*. 2023. – URL https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. – Zugriffsdatum: 2024-07-08

[Williams 2021] Williams, A.: *Hello World! Announcing the Rust Foundation to the World*. 2021. – URL https://foundation.rust-lang.org/news/2021-02-08-hello-world/. – Zugriffsdatum: 2024-06-01

[Yew Contributors a] Yew Contributors: *Docs*. – URL https://yew.rs/docs/getting-started/introduction. – Zugriffsdatum: 2024-07-13

[Yew Contributors b] Yew Contributors: *Yew*. – URL https://github.com/yewstack/yew. – Zugriffsdatum: 2024-07-13

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tool	Verwendung
LaTeX	Textsatz- und Layout-Werkzeug, verwendet zur Erstellung dieses Dokuments
draw.io	Diagramm-Software, verwendet zur Erstellung von Grafiken; Icons und Templates lizensiert unter CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/)
RustRover	IDE von JetBrains, verwendet zur Erstellung des Prototyps
Affinity Designer	Grafikprogramm, verwendet zur Bearbeitung von Screenshots

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Glossar

Bearer Token ermöglicht in HTTP-Requests Zugriff auf Ressourcen, ohne einen kryptografischen Schlüssel einsetzen zu müssen [Jones und Hardt 2012].

Browser siehe *Webbrowser*.

Browser Engine auch *rendering engine*; Teil eines Webbrowsers, der Ressourcen in eine visuelle Abbildung übersetzt [Mozilla Corporation b].

buffer overflow Fehler, bei dem mehr Daten in einen *buffer* geschrieben werden als dieser enthalten kann, wodurch andere Daten überschrieben werden; kann von Angreifer:innen ausgenutzt werden [Stouffer u. a. 2023].

Cargo in Rust entwickelter Paketmanager für Rust [Klabnik und Nichols 2023].

Client Entität, die einen Service abfragt [IEEE 2017].

Common Weakness Enumeration Liste gefundener Softwareschwachstellen [The MITRE Corporation 2023].

Content Security Policy (CSP) durch einen Http-Header umgesetzte Regeln für Ressourcen auf einem Server [Mozilla Corporation a].

Cookie für Zustandsinformationen zu einem User einer Webseite genutzte Datei, die auf dem Clientsystem gespeichert wird [IEEE 2017].

crate Bibliothek für Rust [Blandy u. a. 2021].

Cross-Site Scripting Angriff auf Webanwendungen, bei dem durch Nutzer:inneneingaben Schadcode eingebracht wird [Grassi u. a. 2017].

data race Situation, in der mehrere Threads gleichzeitig den selben Speicherbereich lesen und schreiben [Blandy u. a. 2021].

Document Object Model (DOM) siehe Abschnitt 2.3.

double free Fehler, bei dem ein Programm versucht, die gleiche Speicheradresse zweimal freizugeben [The MITRE Corporation a].

Framework Modelle und Software, die eine Basis für Anwendungen bilden und erweitert werden können [IEEE 2017].

Hashing Verwendung einer Hash-Funktion, die einen String beliebiger Länge unumkehrbar und kollisionsfrei auf einen Schlüssel mit fester Länge abbildet [Grassi u. a. 2017].

Injection Angriff gegen eine Webanwendung, bei dem Befehle eingeschleust werden [Amberg und Schmid 2024].

Macro vordefinierte Sequenz von Anweisungen, die (meist bei der Kompilierung) anstelle eines festgelegten Befehls in ein Programm eingefügt wird [IEEE 1990].

Mid-level Intermediate Representation (MIR) Zwischenzustand bei der Kompilierung von Rust-Quelltext [Matsakis 2016].

Object Relational Mapper Bibliothek, die für die Kommunikation mit einer Datenbank Objektstrukturen auf Relationen abbildet [Starke 2018] .

Phishing Angriff zur Erbeutung von Authentifizierungsdaten durch gefälschte Authentifizierungsaufforderungen, häufig per E-Mail [Grassi u. a. 2017].

Query Builder Modul, das für die Konstruktion von SQL-Anfragen zuständig ist [Diesel Core Team a].

RESTful Webarchitekturen, die eine Client/Server-Struktur haben, nach außen Ressourcen über eine URI zur Verfügung stellen, Standardmethoden wie GET oder POST für die Kommunikation des Client mit Ressourcen verwenden und deren Kommunikation mit Hilfe von Repräsentationen der Ressourcen (wie JSON oder HTML) geschieht; anders als in einer strikten REST-Architektur erfolgt der Kontrollfluss nicht dynamisch über Hypermedia [Starke 2018].

sanitization Entfernen potentiell bösartiger Zeichen aus Eingaben [Stuttard und Pinto 2011].

segmentation fault (Schutzverletzung) Fehler, bei dem ein Prozess versucht, auf eine virtuelle Speicheradresse zuzugreifen, auf die er nicht zugreifen darf [Baun 2020].

semantic versioning (Semantische Versionierung) Regeln für die Nummerierung von Softwareversionen nach dem Schema Major.Minor.Patch, z. B. 1.6.2 [Preston-Werner a].

Server Programm oder System, das einen Service bereitstellt [IEEE 2017].

server-side rendering Verwendung von auf einem Server gespeicherten Anweisungen und Daten, um dynamische Inhalte in einer Webanwendung anzuzeigen [Mozilla Corporation b].

session persistente Interaktion zwischen Client und Server (hier zwischen Nutzer:innen-Browser und Server der Webanwendung) [Grassi u. a. 2017].

Session Token Geheimnis, das einer Sitzung (*session*) zugeordnet ist und anstelle der Authentifizierungsdaten eingesetzt werden kann [Grassi u. a. 2017].

single-page application Webanwendung, die nur ein Dokument von einem Server lädt, das mit Hilfe von JavaScript-APIs ergänzt wird, wenn weitere Inhalte angezeigt werden sollen [Mozilla Corporation b].

Social Engineering Manipulation von Menschen, um Zugriff auf schützenswerte Güter zu erhalten [Stouffer u. a. 2023].

Struct Rust-Datentyp, der mehrere zusammengehörige Werte beinhaltet, die benannt werden (z. B. ein Struct *User* mit den Variablen *username* und *email*) [Klabnik und Nichols 2023].

synchronization primitives einfache Strukturen, um Abläufe in Programmen zu synchronisieren (z. B. Mutex) [Rust Team].

Template Vorlage mit vordefinierten Platzhaltern [IEEE 2017].

TOML (Tom's Obvious Minimal Language) Dateiformat für Konfigurationsdateien, das einfache Lesbarkeit, einfaches Parsen in möglichst vielen Sprachen und eindeutige Übertragbarkeit in eine Hash Table zum Ziel hat [Preston-Werner b].

Trait Verhaltensdefinitionen für einen Typ (z. B. Konstanten oder implementierte Methoden) [Klabnik und Nichols 2023].

use after free auch *dangling pointer*; Fehler, bei dem eine Speicheradresse referenziert oder verwendet wird, nachdem sie freigegeben wurde [The MITRE Corporation b].

web application firewall Firewall für einen Webserver auf Anwendungsebene, die durch ein Regelsystem vor verschiedenen Angriffe schützen kann, ohne dass die zugrundeliegende Anwendung geändert werden muss [Kofler u. a. 2023].

Webbrowser Programm, mit dem Nutzer:innen auf Webseiten zugreifen und mit diesen interagieren können [IEEE 2017].

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original