Hochschule Wismar
University of Applied Sciences Technology, Business and Design
Fakultät für Ingenieurwissenschaften

Master-Thesis

Untersuchung zur Sicherheit von Protokollen mit clientseitiger Passwortverarbeitung und optionaler Mehrfaktorauthentifizierung

Abschlussarbeit zur Erlangung des Grades eines

Master of Engineering

der Hochschule Wismar

Eingereicht am: 21. September 2021

von: Konrad Lohse
geboren am 16.03.1998 in Berlin
Studiengang IT-Sicherheit und Forensik

Abstract

Untersuchung zur Sicherheit von Protokollen mit clientseitiger Passwortverarbeitung und optionaler Mehrfaktorauthentifizierung

Die vorliegende Thesis untersucht Sicherheitseigenschaften von Protokollen zum Account-Management eines Identity-Providers mit Fokus auf Webanwendungen. Diese umfassen folgende Protokollfunktionen: Account-Registrierung, Logins, Sessions, Änderung und Zurücksetzung von Anmeldedaten, Account-Löschung. Beim Einsatz von Passwörtern zur Authentifizierung muss klassischerweise das Backend ein ressourcenintensives Hashing ausführen. Die Thesis untersucht Möglichkeiten und Implikationen zur Verlagerung des Hashings in das Frontend. Hierzu werden verschiedene Protokollvarianten mit clientseitiger Passwortverarbeitung diskutiert sowie ein optimiertes Protokoll konstruiert. Zur Steigerung des Sicherheitsniveaus passwortbasierter Authentifizierung ist eine optionale Mehrfaktorauthentifizierung ergänzbar. Die Thesis betrachtet ihre Integration sowie Eigenschaften und Verwendungsmöglichkeiten symmetrischer und asymmetrischer Verfahren. Auswirkungen der veränderten Protokollabläufe zum Identity-Management werden evaluiert: Die Thesis unterzieht die Protokollentwürfe einem Threat-Modelling zur Beurteilung von Schwachstellen. Die Praktikabilität wird anhand von Tests überprüft, wozu eine prototypische Referenz-Implementierung erfolgt.

Investigation into the Security of Protocols with Client-side Password Processing and Optional Multi-factor Authentication

This thesis investigates security properties of account management protocols for identity providers with a focus on web applications. These include the following protocol functions: Account registration, logins, sessions, changing and resetting of credentials, account deletion. When using passwords for authentication, the backend usually has to perform resource-intensive hashing. The thesis examines the possibilities and implications for relocating hashing to the frontend. For this purpose, different protocol variants with client-side password processing are discussed and an optimized protocol is constructed. To increase the security level of password-based authentication, an optional multi-factor authentication can be added. The thesis considers their integration as well as properties and possible uses of symmetric and asymmetric techniques. Effects of the changed identity management protocol flows are evaluated: The thesis subjects the protocol designs to threat modeling to assess vulnerabilities. Practicability is examined by means of tests, for which a prototypical reference implementation is developed.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Zielsetzung . . . 2

2 Grundlagen **3**
2.1 Ausgangsbasis & Methodik . . . 3
2.1.1 Gegenstand der Untersuchung . . . 3
2.1.2 Angreiferklassifikation . . . 7
2.1.3 Threat-Modelling . . . 9
2.2 Technische Prinzipien & Konzepte . . . 11
2.2.1 Transportverschlüsselung und Zertifikate . . . 11
2.2.2 Föderierte Identitäten und Single-Sign-On . . . 15
2.2.3 Hashing von Passwörtern . . . 16
2.2.4 Tokenbasierte Authentifizierung . . . 22
2.3 Referenz-Protokoll . . . 26
2.3.1 Registrierung und Verifikation neuer Benutzer . . . 26
2.3.2 Anmeldung registrierter Nutzer . . . 30
2.3.3 Sitzungsverwaltung angemeldeter Nutzer . . . 32
2.3.4 Zurücksetzung von Anmeldedaten . . . 33
2.3.5 Änderung von Anmeldedaten . . . 35
2.3.6 Löschung von Accounts . . . 36

3 Methoden clientseitiger Passwortverarbeitung **37**
3.1 Architektonische Probleme . . . 37
3.1.1 Rollenverteilung von Client und Server . . . 37
3.1.2 Verhinderung unsicherer Passwörter . . . 39
3.1.3 Risiko-Bewertung . . . 42
3.2 Clientseitiges Hashing von Passwörtern . . . 45
3.2.1 Angriffe auf Client-Hashing ohne Salt oder Pepper . . . 45
3.2.2 Client-Hashing mit Salt ohne Pepper . . . 48
3.2.3 Client-Hashing mit Salt und Pepper . . . 53
3.2.4 Risiko-Bewertung . . . 57
3.3 Challenge-Response-Verfahren zur Authentifizierung . . . 63
3.3.1 Klassifikation . . . 63
3.3.2 Ausgewählte Protokolle mit bekannten Schwachstellen . . . 65
3.3.3 Salted Challenge Response Authentication Mechanism . . . 69
3.3.4 Risiko-Bewertung . . . 74

4 Mehrfaktor-Authentifizierung **78**
4.1 Einbindung in das Protokoll 78
4.1.1 Client-Peppering via Zweifaktor-Authentifizierung 78
4.1.2 Iterative Mehrfaktor-Authentifizierung 79
4.1.3 Mehr-Augen-Prinzip 81
4.1.4 Ergänzungen zum Referenz-Protokoll 82
4.2 Symmetrische Verfahren 84
4.2.1 Clientseitige Passwort-Generatoren 84
4.2.2 One Time Passwords (OTP) 86
4.2.3 Symmetrische Device-Cookies 93
4.2.4 Risiko-Bewertung 98
4.3 Asymmetrische Verfahren 100
4.3.1 Asymmetrische Device-Cookies 100
4.3.2 Zertifikat-Authentifizierung durch den User-Agent 103
4.3.3 Security-Tokens 105
4.3.4 Risiko-Bewertung 110

5 Realisierung **115**
5.1 Implementierung 115
5.1.1 Umgesetzte Verfahren 115
5.1.2 Datenmodell & Technologien 116
5.1.3 Sicherheit nativer und browserbasierter Applikationen . . . 120
5.1.4 Datengrundlage zur Feststellung schwacher Passwörter . . 122
5.1.5 Rate-Limiting 124
5.2 Tests 127
5.2.1 Benchmarking zur Hashing-Performance von Webbrowsern 127
5.2.2 Vergleich der Hashing-Leistung verschiedener Endgeräte . 130
5.2.3 Benchmarking zum Request-Durchsatz des Servers 132
5.2.4 Einsatzfähigkeit von Security-Tokens mittels FIDO2 134
5.3 Bewertung 136

6 Fazit **139**
6.1 Zusammenfassung 139
6.2 Handlungsempfehlung zu Webapplikationen 139
6.3 Fortführung der Thesis 140
6.4 Ausblick zur technischen Entwicklung 140

7 Thesen **141**

Literaturverzeichnis **142**

Abbildungsverzeichnis **150**

Tabellenverzeichnis **152**

Quelltextverzeichnis **154**

Symbolverzeichnis **155**

Abkürzungsverzeichnis **156**

Glossar **160**

Anlagen **165**

A Übersicht zu vorgestellten Verfahren und betrachteten Problemen **166**

B Dokumentation des Webclients **168**

C Dokumentation der REST-API **173**

D Digitaler Anhang **177**

Selbstständigkeitserklärung **178**

Hinweis

Aus Gründen der besseren Lesbarkeit wird im Text verallgemeinernd die männliche Form verwendet. Diese Formulierungen umfassen gleichermaßen weibliche und männliche Personen.

1 Einleitung

1.1 Motivation

Gegenüber klassischen Desktop-Programmen gewinnen mobile und webbasierte Anwendungen stetig an Bedeutung und lösen diese zunehmend ab. Durch die steigende Verbreitung webbasierter Anwendungen steigt auch die Anzahl an digitalen Identitäten, die ein Nutzer verwalten muss, da für jede Webanwendung potenziell ein eigener Account mit individuellen Zugangsdaten zu pflegen ist. Weil die Anmeldedaten im Netzwerk oder Internet transportiert werden, bedarf es kryptografisch starker Authentifizierungsverfahren zum Schutz dieser Daten. Einem Angreifer bieten sich mehr Möglichkeiten gegenüber hostlokalen Authentifizierungssystemen, wie sie häufig bei Desktop-Programmen verwendet werden. Durch die stetige Verbreitung webbasierter Anwendungen und einer Vielzahl an Anwendern wird der Angriffsvektor weiter vergrößert. Auch wenn zur Implementierung dieser Systeme Standard-Protokolle genutzt werden, kann die Güte der Realisierungen variieren. Betreiber von Webapplikationen ergreifen in der Regel bereits heutzutage wirksame Maßnahmen zum Schutz vertraulicher Anwenderdaten. Eine grundlegende Maßnahme stellt der Einsatz von Transportverschlüsselung, meist via *Transport Layer Security (TLS)*, dar. Allerdings wird dadurch nur der Datentransport geschützt. Zur Verwahrung und Verarbeitung von Anwenderdaten sind weitere Sicherheitsmaßnahmen zu ergreifen, welche jedoch nicht immer dem eigentlich notwendigen Schutzniveau gerecht werden.

Klassischerweise findet zur Nutzer-Authentifizierung eine Kombination aus Nutzerkennung (z. B. E-Mail-Adresse) und Passwort Verwendung. Das Passwort wird insbesondere bei Webapplikationen meist im Klartext an den Server übermittelt und erst dort verarbeitet und ausgewertet. Zum Schutz dieser Übertragung des Klartext-Passwortes wird in der Regel eine Transportverschlüsselung eingesetzt. Eine bereits im Client stattfindende Verarbeitung der Anmeldedaten könnte das erzielbare Sicherheitsniveau erhöhen, da die Übertragung des Passwortwertes entfällt. Zudem ließe sich die Last der Passwort-Verarbeitung vom Server zum Client verlagern. Das Passwort als Authentifizierungsmerkmal befindet sich in einem Spannungsfeld aus Anforderungen, einerseits an möglichst hohe Komplexität, sowie andererseits an Alltagstauglichkeit und Erlernbarkeit für den Nutzer. Zur Entschärfung des Passwortes als alleiniges Merkmal werden zum Nachweis der Identität eines Nutzers zunehmend auch Verfahren zur *Mehrfaktor-Authentifizierung (MFA)* implementiert. Hierbei werden weitere Nutzermerkmale in den Authentifizierungsprozess eingebettet, sodass der Nachweis oder das Fäl-

schen einer einzigen korrekten Merkmalsausprägung nicht mehr ausreichend ist. Allerdings senkt *MFA* die Nutzerfreundlichkeit der Authentifizierungsprozesse, weshalb der Einsatz entsprechender Techniken meist noch nicht als Standardeinstellung bei Webapplikationen etabliert ist. Häufig findet in der Praxis aus verschiedenen Gründen auch keine Implementierung derartiger Verfahren statt. Durch Integration weitergehender Maßnahmen könnte eine mehrschichtigen Sicherheitsarchitektur aufgebaut und somit die Sicherheit applikationsspezifischer Vorgänge der Nutzer-Authentifizierung gesteigert werden.

1.2 Zielsetzung

Betrachtungsgegenstand dieser Thesis sind Nutzerverwaltung und Identity Management von Webapplikationen. Verschiedene Varianten von Verfahren der clientseitigen Passwortverarbeitung und Mehrfaktorauthentifizierung werden vorgestellt und diskutiert. Die erarbeiteten Verfahren sollen prototypisch implementiert werden. Dies erfolgt in einer exemplarischen Webanwendung mit Client- und Server-Komponente. Es gilt neben der Realisierbarkeit und Nutzbarkeit vor allem die Sicherheit der Verfahren gegenüber Angriffen zu untersuchen. Zur Beurteilung von Risiken durch Angriffe auf das System wird ein Threat Modelling durchgeführt. Zu den vorgestellten Sicherheitsmaßnahmen werden Angriffsszenarien skizziert, welche die Angemessenheit und Tauglichkeit der Maßnahmen im jeweiligen Kontext herausstellen.

Der Schwerpunkt der clientseitigen Passwortverarbeitung verfolgt den Ansatz, dass die Server-Komponente zu keinem Zeitpunkt Kenntnis über das Klartext-Passwort eines Nutzers besitzen soll. Es gilt die Möglichkeiten und Grenzen einer Passwortverarbeitung, die ausschließlich in der Client-Komponente das Klartext-Passwort verwendet, zu evaluieren. Die Sicherheit des Authentifizierungsvorganges soll durch diese Maßnahme nicht beeinträchtigt und falls möglich gesteigert werden. Der Server soll durch die clientseitige Verarbeitung entlastet werden.

Zur Untersuchung von Verfahren zur *MFA* werden verschiedene software- und hardwarebasierte Techniken mit ihren individuellen Vor- und Nachteilen vorgestellt. Ziel ist eine nutzbare und alltagstaugliche Realisierung eines Systems mit optionaler *MFA*. Dabei werden die Techniken der clientseitigen Passwortverarbeitung und der *MFA* kombiniert. Es soll eine Implementierung entstehen, die beide Aspekte unter Gesichtspunkten von Schutz der Nutzer, Skalierbarkeit im Betrieb, Nutzerfreundlichkeit und Sicherheit der eingesetzten Verfahren vereint.

2 Grundlagen

In den nachfolgenden Kapiteln werden verschiedene Protokollvariationen diskutiert. Als Ausgangsbasis erfolgt in diesem Kapitel die Definition eines Referenz-Protokolls mit serverseitiger Passwortverarbeitung und ausschließlicher *Einfaktor-Authentifizierung (1FA)* anstatt *MFA*. Gegenüber diesem Protokoll gilt es Verbesserungen vorzunehmen. Hierzu bedarf es zunächst einer thematischen Einordnung der zu realisierenden Applikation und der damit verbundenen Zielsetzung. Dazu werden nachfolgend Architektur und Anforderungen an die Anwendung bestimmt. Anschließend werden mögliche Risiken und Methoden zur Bewertung vorgestellt. Außerdem werden weitergehende technische Aspekte im Kontext clientseitiger Passwortverarbeitung sowie *MFA* betrachtet.

2.1 Ausgangsbasis & Methodik

2.1.1 Gegenstand der Untersuchung

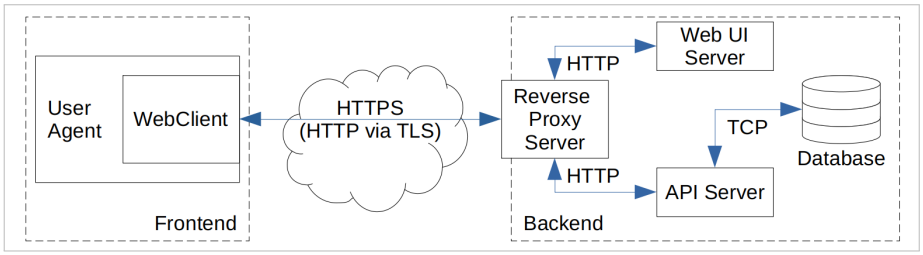


Abbildung 2.1: Grundlegender Aufbau und Kommunikationspfade der Applikation
Quelle: Eigene Darstellung

Untersuchungsgegenstand dieser Thesis sollen Webapplikationen sein. Sie sind durch eine Client-Server-Architektur und der Verwendung von *Hypertext Transfer Protocol (HTTP)* als primäres Übertragungsprotokoll gekennzeichnet. Als Teil der Ausarbeitung soll eine Implementierung entstehen, deren Architektur in Abbildung 2.1 skizziert ist. Die Kommunikation zwischen Frontend und Backend über das Internet wird mittels Transportverschlüsselung abgesichert, welche Inhalt von Kapitel 2.2.1 ist. Wenn das Backend als gesicherter Bereich betrachtet wird, ist mitunter auf eine interne Transportverschlüsselung zugunsten der Leistungsfähigkeit verzichtbar. Die Absicherung wird dann durch starken Zugriffsschutz anstatt durch Verschlüsselung erzielt. Dieser Zugriffsschutz ist beispielsweise gegeben, wenn sämtliche Backend-Komponenten isoliert betrieben werden und nur über lokale Netzwerk-Sockets miteinander kommunizieren können. Falls dies nicht zu gewährleisten ist, gilt es auch interne Kommunikationsstrecken zwischen Backend-Komponenten via *TLS* zu schützen. Ein besonders hohes Schutzniveau

bietet Verschlüsselung via *TLS* in Kombination mit starkem Zugriffsschutz, was allerdings auch die Systemkomplexität erhöht.
Der *Reverse-Proxy-Server* ist als zentraler Zugangspunkt zum Backend besonders schützenswert. Zudem terminiert er die *TLS*-Verbindung zum Client. Im Frontend führt der *User Agent (UA)* den *Webclient* aus. Als *UA* wird das Programm bezeichnet, welches die Client-Komponente einer Webapplikation ausführt. Die bekannteste Form eines *UA* ist der Webbrowser, wobei hierzu generell Programme zählen, die als Client einer *HTTP*-Datenübermittlung fungieren. Zur browserbasierten Nutzung der Applikation wurde ein grafisches *User Interface (UI)* implementiert. Das Web-UI wird vom Backend als Server-Dienst bereitgestellt. Der Informationsaustausch zwischen Frontend und Backend erfolgt über eine *Application Programming Interface (API)*. Die *API* wird vom Backend zur Verfügung gestellt und durch kompatible *UAs* konsumiert. Der zur Bereitstellung der *API* zuständige Server leitet die Informationsverarbeitung durch das Backend und interagiert hierzu auch mit der *Datenbank (DB)*. Die *DB* dient als alleinige Persistenzschicht des Backends.

Bei den *UAs* müssen auch leistungsschwache Endgeräte bedacht werden, wobei hohe Wartezeiten durch rechenintensive Operationen zu vermeiden sind. Zudem sollen bei mobilen Geräten Energieverbrauch und Volumen der Datenübertragung möglichst gering gehalten werden. Beim Einsatz von Webbrowsern als *UA* muss beachtet werden, dass die Webapplikation auf die via Browser-*Sandbox* bereitgestellten Schnittstellen beschränkt ist.

Die Server-Software soll konzeptionell eine gute Performance bieten, was vor allem anhand des Durchsatzes parallel verarbeitbarer Client-Anfragen messbar ist. Eine Leistungssteigerung lässt sich architektonisch zum einen durch eine Lastverschiebung vom Server hin zum Client erzielen, sodass mit gleichen Server-Ressourcen mehr Client-Anfragen in gleicher Zeit bearbeitbar sind. Wie oben erläutert, besteht dann die Gefahr einer Überlastung leistungsschwacher Endgeräte.
Die Leistungssteigerung ist aber auch über eine Server-Implementierung mit hoher Skalierbarkeit erreichbar. Durch die Skalierung werden mehrere Instanzen der gleichen Server-Software im Verbund betrieben. Diese Instanzen verarbeiten zusammen als Server-Cluster die Client-Requests, wobei der Durchsatz durch Verteilung der Requests auf die einzelnen Instanzen gesteigert wird. Diese Skalierbarkeit erreicht ihr Supremum, sobald der Aufwand zur Koordination des Clusters soweit angestiegen ist, dass der Gesamtdurchsatz durch das Hinzufügen weiterer Instanzen gesenkt würde, anstatt ihn zu steigern. Der Overhead zur Koordination entsteht durch geteilten Speicher zwischen den Instanzen, welcher als Zustand bezeichnet wird. Eine zustandslose Anwendung besitzt entsprechend eine

unbeschränkte Skalierbarkeit. Konzeptionell soll die Implementierung möglichst zustandsarm sein, um eine hohe Skalierbarkeit zu ermöglichen.

Die zu untersuchende Architektur-Komponente einer Webapplikation mit integrierter Nutzerverwaltung kann als *Identity Provider (IDP)* bezeichnet werden. Der *IDP* erfüllt seine Aufgabe durch die Kommunikation zwischen Front- und Backend. Dieser fest definierte Informationsaustausch wird im Folgenden als *IDP*-Protokoll bezeichnet. Der *IDP* soll folgende Aufgaben erfüllen:

1. **Registrierung** neuer Benutzer am System und Verifikation des Accounts
2. **Anmeldung** registrierter Nutzer am System und Vergabe von Berechtigungen (Authentifizierung und Autorisierung)
3. **Sitzungen** für angemeldete Nutzer ausstellen und verwalten
4. **Änderung** der Anmeldedaten im Kontext einer gültigen Sitzung
5. **Zurücksetzung** von Anmeldedaten bei Verlust (ohne gültige Anmeldung)
6. **Löschung** des Accounts durch den Nutzer

Diese Aufgaben werden im Folgenden als *IDP*-Protokollfunktionen bezeichnet. Ihre Zusammenhänge stellt Abbildung 2.2 dar. Die Ausgestaltung der Funktionsinhalte ist Gegenstand von Kapitel 2.3. Die Betrachtung legt den Schwerpunkt auf Prozesse der Authentifizierung und weniger auf Konzepte der Autorisierung.

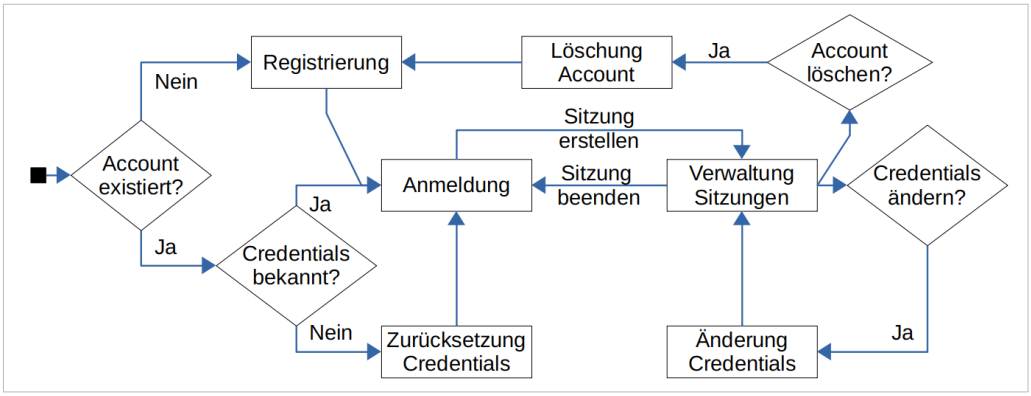


Abbildung 2.2: Zustandsübergänge der Protokollfunktionen
Quelle: Eigene Darstellung

Zum Nachweis der Identität des Nutzers werden bei Registrierung nutzerspezifische Merkmalsausprägungen im System hinterlegt. Diese werden zur Authentifizierung abgefragt und können aus folgenden Merkmalskategorien stammen:

- **Wissen** um ein Geheimnis: z. B. Passwort oder Passphrase
- **Besitz** eines einmaligen Objektes: z. B. analoger oder digitaler Schlüssel
- **Inhärenz** des zu Authentifizierenden: z. B. biometrischer Fingerabdruck

Diese Thesis legt den Fokus auf Merkmale der Kategorien Wissen und Besitz. Besonderheiten der Inhärenz werden nicht untersucht. Die Betrachtung zur Umsetzbarkeit der Merkmale als Faktoren zur Authentifizierung wird in folgende Dimensionen unterteilt:

- **Security**: Durch Einsatz des Faktors erzielbares Sicherheitsniveau
- **Usability**: Nutzern auferlegter Aufwand zum Nachweis des Merkmals
- **Deployability**: Aufwand und Möglichkeiten zur Etablierung des Faktors im System und bei der Zielgruppe der Nutzer

Beispielsweise ist die Security des Merkmals Passwort von der Länge abhängig, wobei die Usability mit steigender Länge sinkt. Dafür ist die Deployability sehr gut, da das Merkmal auf beliebigen Endgeräten mit Tastatureingabe nutzbar ist. Im Gegensatz dazu erzielen Hardware-Tokens, siehe Kapitel 4.3.3, eine hohe Security sowie Usability, erzeugen aber einen Kostenaufwand und sind nur auf Endgeräten mit geeigneten Schnittstellen sowie passender Software-Unterstützung nutzbar.

Der Merkmalstyp Passwort konnte ursprünglich einzig der Kategorie Wissen zugeordnet werden. Diese Einordnung berücksichtigt allerdings nicht den Trend der Passwort-Manager: Anstatt sich viele verschiedene Passwörter zu merken, kann ein Nutzer diese in einem Passwort-Manager aufbewahren. Dadurch wird das Gedächtnis des Nutzers entlastet und die Passwort-Sicherheit kann gesteigert werden, da nun Länge und Komplexität der Passwörter nicht mehr durch die Merkfähigkeiten des Nutzers begrenzt sind. Der Nutzer muss sich lediglich ein einzelnes, starkes Main-Passwort zum Schutz der gespeicherten Passwörter merken. Diesen Ansatz vertieft Kapitel 4.2.1. Allerdings ist diese Verwendung von Passwörtern der Kategorie Besitz zuzuordnen. Deshalb werden im Folgenden zur Unterscheidung die Begriffe Passwortkenntnis und Passwortbesitz eingeführt.

Vorrangig soll die Kommunikation zwischen Mensch und Maschine gegenüber einer reinen *Machine-to-Machine (M2M)*-Kommunikation betrachtet werden. Mit ihrer ursprünglich konzipierten, begrenzten Länge sind Passwörter zur Verwendung als Wissensmerkmal durch Menschen besonders geeignet. Eine Maschine kann sich ebenfalls durch Passwortbesitz in ihrem Speicherzugriffsbereich authentifizieren. Jedoch wären stattdessen auch stärkere Merkmale der Kategorie Besitz verwendbar, die Schlüsselmaterial mit höherer Entropie einsetzen. Hierzu zählen z. B. Zertifikate, welche in Kapitel 2.2.1 erläutert werden.

2.1.2 Angreiferklassifikation

Um die Bedrohungen, denen der zu entwerfende *IDP* ausgesetzt sein kann, zu erfassen, sind zum Threat Modelling zunächst mögliche Angreifer zu klassifizieren. Parallel erfolgt auch eine Aufzählung von Maßnahmen, die in der Ausarbeitung gegen die entstehenden Bedrohungen entwickelt werden.

Script-Kiddies

Sie besitzen geringe technische Kenntnisse, können aber mithilfe entsprechender Werkzeuge auf das System einwirken. Eine typische Technik ist der *Distributed Denial Of Service (DDOS)*-Angriff, welcher das Backend mit Anfragen überlastet. Da der Server unauthentifiziertes Passwort-Hashing vollführen muss, kann ein Angreifer hierüber Last erzeugen. Ein Rate-Limiting kann den Angriff abmindern, siehe Kapitel 5.1.5. Zusätzlich wird die Hauptlast des Hashings vom Server zum Client verlagert, siehe Kapitel 3.
Script-Kiddies könnten auch gängige Passwörter für verschiedene Nutzerkennungen durchprobieren. Der *IDP* muss die Wahl schwacher Passwörter unterbinden, siehe Kapitel 3.1.2.

Hacker

Der Hacker als generischer Angreifertypus besitzt ein hohes technisches Verständnis. Er kann Schwachstellen in *IDP*-Protokoll sowie -Implementierung finden und zu Exploits kombinieren. Es wird davon ausgegangen, dass ein Hacker Aufbau, Quellcode und Konfiguration des Systems kennt. Die Sicherheit von Applikation und Datenbestand darf nur von den eingesetzen Verfahren und der Geheimhaltung des zugehörigen Schlüsselmaterials abhängen.
Ein Hacker kann anstatt der *IDP*-Applikation auch ein Client-System kompromittieren und z. B. einen *Keylogger* installieren. Für dieses Szenario ist die *MFA* mit getrennten Geräten vorgesehen, siehe Kapitel 4. Der Hacker soll dadurch keinen eigenen Login reproduzieren können. Jedoch kann er die Sitzung auf dem kompromittierten System übernehmen. Damit ist er in der Lage u. a. Nutzerdaten einzusehen und eine Impersonation-Attack auszuführen, also unter der Identität des Opfers zu agieren. Ein Schutz hiervor ist nicht durch die in dieser Thesis thematisierten Verfahren zu leisten und daher kein Teil des Threat Models. Das Risiko ist zu akzeptieren.
Ein Hacker kann auch das Backend angreifen. Die Absicherung von Server-Infrastruktur sowie separaten Administrationszugängen ist kein Teil der Betrachtung. Der Einfluss einer Server-Kompromittierung auf die Clients wird in Kapitel 5.1.3

diskutiert.
Der Angreifer kann auch gegen Nutzer vorgehen. Besitzt der Hacker aus vorherigen Hacks eine Zusammenstellung von Nutzername-Passwort-Tupeln, so kann er dazu eine Credential-Stuffing-Attacke starten. Dabei probiert er unter der Annahme, dass Nutzer ähnliche Credentials wiederverwendet haben, Abwandlungen der Credential-Tupel bei anderen Diensten, um Zugriff zu erlangen. Dies ist erfolgversprechender als ein Brute-Force-Angriff. Alternativ kann er Phishing betreiben.

Phisher und Spear-Phisher

Ein Phisher versucht per Social Engineering Zugang zu möglichst vielen Accounts von Nutzern der Anwendung zu erlangen. Dazu versendet er z. B. Phishing-Mails, die Nutzer durch trügerische Behauptungen dazu animieren, dem Angreifer Zugang zu ermöglichen. Dies kann z. B. durch Login auf einer Phishing-Webseite geschehen, die optisch der originalen entspricht. Daten potenzieller Opfer hat er z. B. aus *Data Breaches* oder aus öffentlich-einsehbaren Daten per *Open Source Intelligence (OSINT)* gesammelt.
Im Gegensatz zum Phisher hat ein Spear-Phisher ein konkretes Opfer im Visier und kann anhand gesammelter Informationen zum Opfer einen maßgeschneiderten Phishing-Angriff ausführen. Da nur eine oder wenige Personen zur Zielgruppe gehören, kann der Spear-Phisher seine Ressourcen stärker konzentrieren. Er könnte z. B. darüber hinaus auch mit leistungsfähiger Hardware und technischer Expertise Rainbow-Tables für spezifische Accounts aufbauen, siehe Kapitel 2.2.3.
Der *IDP* soll Nutzer vor Phishing schützen, indem Logins auf Fake-Websites unterbunden werden, siehe Kapitel 4. Das Risiko einer Credential-Weitergabe des Opfers an den Angreifer muss für Wissens-Merkmale akzeptiert werden. Aus Besitz-Merkmalen lassen sich extraktionsfreie Faktoren konstruieren, sodass eine Weitergabe des Schlüsselmaterials verhinderbar wird, siehe Kapitel 4.3.

Man In The Middle und Eavesdropper

Diese Angreifer sind in den via *TLS* gesicherten Kanal zwischen Client und Server eingebrochen, siehe Kapitel 2.2.1. Dies kann durch Protokoll-Fehler (z. B. eine *Downgrade-Attacke*), Implementierungsfehler (z. B. Heartbleed-Bug) oder aufgrund von Fehlkonfiguration erreicht werden. Ein *Eavesdropper* belauscht die Kommunikation im Klartext. Ein *Man In The Middle (MITM)* kann Daten abfangen, manipulieren oder auch einspeisen. Falls der Angreifer nicht die Klartext-Übertragung einsehen kann, ist immer noch eine Replay-Attacke denkbar. Dabei legt der Angreifer aufgezeichnete Kommunikation erneut am Server vor, um die

Identität des Opfers vorzuspielen.
Das *IDP*-Protokoll soll verhindern, dass ein *MITM* aus abgefangenen Daten Credentials zum Zugriff auf die Applikation bestimmen kann. Hierzu werden Challenge-Response-Verfahren etabliert, siehe Kapitel 3.3. Das Threat Model sieht vor, dass nur die Credentials vor einem *MITM* geschützt werden. Ein erfolgreicher *MITM* kann damit trotzdem Sitzungen vollständig übernehmen. Dies wird akzeptiert, da die Mitigation hiergegen der Etablierung eines gesicherten Kanals durch das *IDP*-Protokoll entsprechen würde. Damit wäre eine architektonische Dopplung zu *TLS* gegeben, die nicht zielführend sein kann.

Angreifer von Innen

Dieser Angreifertyp besitzt im Gegensatz zu den vorherigen legitimen Zugriff auf die Applikation, z. B. in der Rolle eines Mitarbeiters oder Dienstleisters. Der Schutz vor Manipulationen an Applikation oder Datenbestand ist eine organisatorische Maßnahme mit technischen Umsetzungsmöglichkeiten, welche jedoch kein Teil der Betrachtung sind. Wenn der Angreifer über Zugriff zur Datenbank verfügt, kann er Nutzerdaten abgreifen. Eine Impersonation-Attacke soll jedoch nur mit erhöhtem Aufwand möglich sein, siehe Kapitel 2.2.3.

2.1.3 Threat-Modelling

In Kapitel 3 und 4 folgt der Entwurf verschiedener Protokolle in abgewandelten Varianten. Diese besitzen abweichende Sicherheitsniveaus in Folge unterschiedlich starker Anfälligkeiten für bestimmte Bedrohungen. Um das Risiko, das durch den Einsatz der Protokolle entsteht, vergleichbar zu machen, wird ein Threat-Modelling betrieben. Techniken zum Threat-Modelling erlauben es, Einschätzungen zur Kritikalität anhand definierter Metriken abzugeben. Nachfolgend wird der Prozess zur Auswahl eines geeigneten Verfahrens beschrieben.

Um mögliche Risiken bewerten zu können, wird ein Threat-Modelling betrieben, welches auf die Angreiferklassifikation aus Kapitel 2.1.2 aufbaut. Etablierte Verfahren zum Threat-Modelling beziehen sich typischerweise auf existierende Applikationen und evaluieren ihre Schwächen im Einsatzkontext. Beispielsweise ist über das Framework *Common Vulnerability Scoring System (CVSS)* jeder Sicherheitslücke ein Scoring zum Schweregrad zuordenbar, welches sich aus verschiedenen Einzelbewertungen zusammensetzt. Darin fließen auch zeitliche und kontextuelle Aspekte ein, z. B. ob bereits Exploit-Code existiert oder ob nur ein Workaround anstatt einer Problemlösung möglich ist.[FIR19, S. 12 ff.] In dieser Thesis sollen jedoch keine bereits implementierten Applikationen, sondern Pro-

tokoll-Designs untersucht werden. Damit ist die Methodik nicht für diesen Zweck geeignet.
Gleiches gilt für das Framework *Process for Attack Simulation and Threat Analysis (PASTA)*, bei dem ausgehend von Unternehmenszielen in einem siebenschrittigen Prozess Angriffsoberflächen und Risiken analysiert werden.[McG13] Den gesamten *PASTA*-Prozess nachzuvollziehen, würde nur Einschränkungen und Annahmen bezüglich der Use-Cases schaffen. Jedoch sollen in dieser Thesis die Einsatzmöglichkeiten der Protokolle so allgemein und abstrakt wie möglich gehalten werden, um ein breites Spektrum an Nutzungsszenarien zu ermöglichen. Lediglich der *PASTA*-Prozessschritt zur Schwachstellen-Analyse ist auf das Protokoll-Design übertragbar. Nach *PASTA* können hierfür u. a. das bereits erläuterte *CVSS* oder auch Threat-Trees eingesetzt werden. Letztere bewerten eine Schwachstelle in Abhängigkeit der zur Ausnutzung notwendigen Vorbedingungen. Ein Threat-Modelling, das nur diese Eigenschaft untersucht, würde zu kurz greifen. Daher wurden notwendige Vorbedingungen als Dimension für das durchgeführte Threat-Modelling berücksichtigt.

Die Technik *Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of Privilege (STRIDE)* ordnet Threats in die akronymbildenden Kategorien ein. Zu jedem Threat wird vermerkt, welche Kategorien dieser betrifft. Dadurch können überblicksmäßige Aussagen über die Verletzung der Schutzziele Authentizität, Autorisierung, Integrität, Nicht-Abstreitbarkeit, Verfügbarkeit und Vertraulichkeit getätigt werden.[DJPJ05] Allerdings erfolgt keine qualitative Einschätzung zum Schadenspotenzial. Damit ist die Metrik ungeeignet, wie gewünscht, Threats zu vergleichen.
Währenddessen beurteilt *Damage, Reproducibility, Exploitability, Affected-users, Discoverability (DREAD)* Threats anhand der namensgebenden Aspekte:

- **D**amage: Der durch einen Angreifer verursachbare Schaden
- **R**eproducibility: Schwierigkeit zum Ausnutzen der Schwachstelle
- **E**xploitability: Notwendige Vorbedingungen
- **A**ffected-users: Größenordnung betroffener Nutzer
- **D**iscoverability: Schwierigkeit zum Auffinden der Schwachstelle

Die Discoverability ist kritisch zu sehen, da sie zur Denkweise „security through obscurity„ verleitet. Insbesondere für Protokoll-Designs darf diese Eigenschaft nicht erheblich sein, weshalb sie nicht zu berücksichtigen ist.
In der Variante DREAD-D (ohne Discoverability) ist die Methodik zur Modellierung von Protokoll-Schwächen geeignet. Zur Beurteilung einer Bedrohung wird den vier Metriken ein Score zwischen eins und fünf zugeordnet, wobei ein hoher

Wert einem schwerwiegenden Problem entspricht. Je Threat werden die Metrik-Scores bestimmt und zu einem Scoring-Wert aufsummiert.[McG13] Anstatt einer einfachen Summierung wäre auch eine unterschiedliche Gewichtung der Metrik-Scores zueinander möglich. Diese Entscheidung muss jedoch in Abhängigkeit der Sicherheitsanforderungen einer konkreten Applikation sowie ihren Use-Cases getroffen werden. Deshalb wird für die abstrakten Protokoll-Entwürfe mit einer unverzerrten Gewichtung gearbeitet. Aus dem Scoring wird eine Risiko-Einschätzung abgeleitet. Tabelle 2.1 zeigt die an [Ala18] angelehnte Bewertungsskala, welche für das Threat-Modelling zum Einsatz kommt.

Tabelle 2.1: Einteilung der Risikobewertung anhand ermittelter Score-Werte

Summierter Score	4 – 9/20	10 – 14/20	15 – 20/20
Risikobewertung	Niedrig	Mittel	Hoch

2.2 Technische Prinzipien & Konzepte

Die folgenden Betrachtungen erfordern technische Grundlagen, die in diesem Abschnitt beschrieben werden.

2.2.1 Transportverschlüsselung und Zertifikate

Transportverschlüsselung dient dem Schutz einer Kommunikationsübertragung vor Einflussnahme durch *MITM*. Sie kann u. a. über Protokolle wie *Transport Layer Security (TLS)*, *IP Security (IPSec)* oder auch *Secure SHell (SSH)* umgesetzt werden.
Bei modernen Webapplikationen gilt in den letzten Jahren zunehmend der Einsatz von *Secure HTTP (HTTPS)* als obligatorischer Sicherheitsstandard: So erlauben moderne Webbrowser nur innerhalb eines sicheren Kontextes Zugriff auf bestimmte Features, z. B. die *API* zur Positionsbestimmung. Für Webbrowser ist der sichere Kontext einer Webapplikation durch die ausschließliche Verwendung von *HTTPS* zur Kommunikation mit dem Backend gegeben.[Wes16]
Das Protokoll *HTTPS* bietet der aufrufenden Applikation eine *HTTP*-Schnittstelle und verwendet zum Schutz des Datentransports das *TLS*-Protokoll. Mittels *TLS* können die Schutzziele der Integrität, Vertraulichkeit und Authentizität einer Datenübertragung umgesetzt werden. Zudem lässt sich durch die transparente Arbeitsweise von *TLS* das Protokoll einer Webapplikation von *HTTP* auf *HTTPS* umstellen, ohne die Datenübertragungen der Applikation inhaltlich anpassen zu müssen. Aufgrund dieser leichten Adaptierbarkeit können auch bestehende Web-

anwendungen durch *TLS* abgesichert werden.

TLS verwendet den *Public Key Infrastructure (PKI)*-Ansatz, der zum Besitznachweis einer digitalen Entität Zertifikate im X.509-Format nach [BSP+08] nutzt. Im Kontext von Webapplikationen wird über diese Zertifikate typischerweise der Besitz einer Domain innerhalb des Gültigkeitszeitraums nachgewiesen.[1] Die hierzu notwendigen, in einem Zertifikat enthaltenen Angaben zeigt Abbildung 2.3 mit hinreichender Detailtiefe für die Betrachtung. Gegenstand der Zertifizierung sind die betreffenden Domainnamen des Domain-Inhabers. Dieser nutzt einen asymmetrischen Schlüssel bestehend aus Public- und Private-Key. Der Public-Key wird in die Zertifikat-Angaben aufgenommen und zur Verifikation des Zertifikats verwendet. Mithilfe des Private-Key werden die Angaben durch den Inhaber signiert. Wenn der Inhaber sich das Zertifikat vollkommen eigenständig ausstellt, ist es ein selbstsigniertes Zertifikat. Sein Einsatz bietet Integrität und Vertraulichkeit aber keine Authentizität, da keine Autorität das Zertifikat als vertrauenswürdig anerkannt hat. Um Vertrauenswürdigkeit herzustellen, werden die Angaben als *Certificate Signing Request (CSR)* der zuständigen *Certificate Authority (CA)* vorgelegt. Die *CA* prüft die Angaben im *CSR*. Daraufhin erstellt sie ein Zertifikat, indem sie die Angaben vom Inhaber mit *CA*-Angaben anreichert. Dies umfasst u. a. den *CA*-Public-Key zur Offline-Verifizierbarkeit, sowie den Verweis zum *CA*-Zertifikat zur Dokumentation. Aus diesen Angaben wird das Nutzer-Zertifikat erzeugt, indem die *CA* die Angaben mit ihrem Private-Key signiert. Abschließend wird das erstellte Zertifikat dem Domain-Inhaber zur Verwendung zur Verfügung gestellt. Dieser Prozess erlaubt eine Zertifikatsgenerierung unter Geheimhaltung der Private-Keys von *CA* und Inhaber.

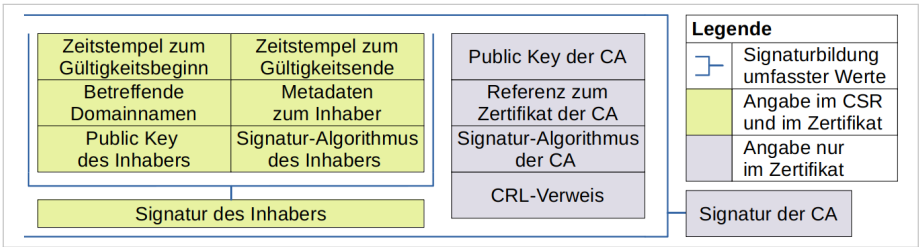


Abbildung 2.3: Modell zu Inhalt und Zusammensetzung eines X.509-Zertifikats
Quelle: Eigene Darstellung

Wie in Abbildung 2.4 skizziert, können *CAs* mehrschichtig angeordnet werden, sodass eine Hierarchie an Institutionen und Zuständigkeitsbereichen entsteht. Beispielsweise kann eine Institution in der Rolle einer *CA* zweckgebundene Sub-*CAs* zu *TLS*-, Mail- und Software-Signatur eröffnen. Die Wurzel der entstehenden Baumstruktur wird als Root-*CA* bezeichnet. Die Baumblätter werden Signing-

[1] Weitere Verwendungsmöglichkeiten werden in Kapitel 4.3.2 und 4.3.3 thematisiert.

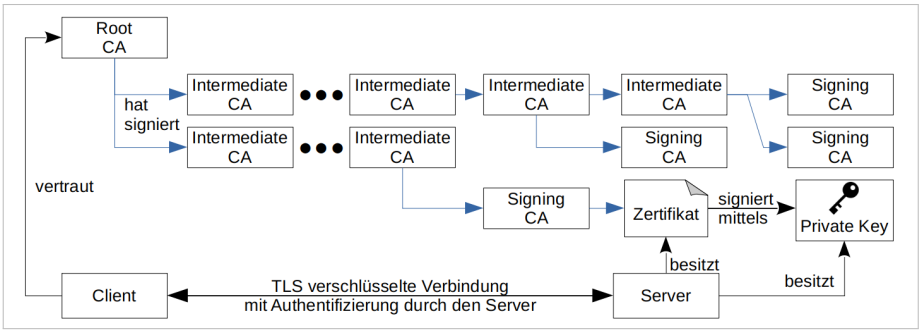


Abbildung 2.4: Beispiel zu Aufbau und Funktionsweise einer PKI-Architektur
Quelle: Eigene Darstellung

CA genannt und dienen der Ausstellung von Nutzer-Zertifikaten, welche als Endinstanzen nicht in Form einer *CA* agieren dürfen. Die zwischenliegenden Schichten beinhalten Intermediate-*CAs*. Die Root-*CA* bildet den Vertrauensanker der Architektur, wobei ein Teilnehmer auch mehreren Root-*CAs* vertrauen kann. Um geschützt via *TLS* kommunizieren zu können, müssen die Teilnehmer der gleichen, zuständigen Root-*CA* vertrauen. Außerdem müssen die Zertifikate der Teilnehmer und rekursiv die der Aussteller gültig sein. Sonst ist die Zertifikatskette gebrochen und kein Schutz vor *MITM*-Angriffen gegeben. Lediglich ein passives Eavesdropping wird dann durch die etablierte Verschlüsselung unterbunden.[SPS11]
Wird eine *CA* kompromittiert, kann der Angreifer vermeintlich vertrauenswürdige Zertifikate ausstellen, mitunter auch über den Zuständigkeitsbereiches der *CA* hinaus. Deshalb muss den seit Kompromittierung durch die *CA* ausgestellten Zertifikaten das Vertrauen entzogen werden. Durch Aufnahme eines Zertifikats auf die zugehörige *Certificate Revocation List (CRL)* wird das Vertrauen über den Nachweis des Entitätsbesitzes entzogen. Verifizierende Instanzen laden die in einem Zertifikat referenzierte *CRL*-Datei herunter und halten sie im Cache. Dadurch können sie prüfen, ob das vorgelegte Zertifikat nicht enthalten ist. Eine sicherere Alternative mit besserer Aktualität bietet das *Online Certificate Status Protocol (OCSP)*, bei dem die verifizierende Instanz vor jeder Verwendung einen Validierungsdienst zum Sperrstatus des Zertifikats befragen kann.[SMA+13]

Im Kontext der Zertifizierung von Domainbesitz besteht eine architektonische Dopplung zwischen dem *TLS*- und *Domain Name System (DNS)*-Protokoll. Über *DNS*-Records werden Zuordnung und Besitz von Domains bestimmt, was u. a. zum Routing für das *Internet Protocol (IP)* verwendet wird. Die Zertifikatsinfrastruktur baut darauf mit dem oben erläuterten Verifikationssystem auf. Ein Angreifer könnte einen *DNS*-Record dahingehend manipulieren, dass er auf einen angreiferkontrollierten Server verweist. Für die übernommene Domain könnte er sich dann durch eine *CA* ein *TLS*-Zertifikat ausstellen lassen. In dieser Position kann der Angreifer als *MITM* zwischen Clients und Dienstbetreiber fungieren.

Trotz intakter *TLS*-Zertifikate kann er die weitergeleitete Kommunikation unverschlüsselt mitlesen sowie manipulieren. In Bezug auf Webanwendungen hängt der Schutz durch Einsatz von *TLS* damit transitiv auch von der Sicherheit der *DNS*-Realisierung ab.
Als Gegenmaßnahme kann durch den Serverbetreiber *HTTP Public Key Pinning (HPKP)* eingesetzt werden. Dadurch lehnen kompatible *UAs* gültige Zertifikate mit Public-Keys ab, die von den hinterlegten Public-Keys abweichen.[EPS15] Eine fehlerhafte Handhabung kann jedoch zu einem *Denial Of Service (DOS)* führen, weshalb das Verfahren nur durch wenige *UAs* umgesetzt wird.[Ris16] Eine Alternative bietet das Verfahren „Certificate Transparency“, bei dem ausgestellte Zertifikate zur Feststellung von Abweichungen in einer öffentlichen Registry zu vermerken sind. Die Durchsetzung erfolgt, indem *UAs* nur Zertifikaten vertrauen, die in der Registry aufgeführt sind.[LLK13]

Das Protokoll *TLS* gilt in den aktuellen Versionen 1.2 und 1.3 unter Einsatz bestimmter Cipher-Suites[2] als geeignet, um einen sicheren Übertragungskanal zu bieten.[BSI21] Dies setzt jedoch voraus, dass Implementierung und Konfiguration des Protokolls bei allen Teilnehmern fehlerfrei sind. Zudem erfordern die genannten architektonischen Schwächen, dass zusätzlich in der Applikation Sicherheitsvorkehrungen getroffen werden. Insbesondere bei Authentifizierungsprozessen können auf Applikationsebene weitere Maßnahmen ergriffen werden, da Charakteristika eines Nutzermerkmals clientseitig und serverseitig bekannt sind: z. B. das Klartext-Passwort im Client und der zugehörige Hashwert im Server. Durch Anwendung entsprechender Verfahren kann damit eine Passwort-Übertragung an den Server vermieden werden, was Kapitel 3 thematisiert.

Über *Mutual TLS (mTLS)* lässt sich eine wechselseitige Authentifizierung mittels *TLS* umsetzen. Dabei weist regulär der Server gegenüber dem Client seine Identität mittels Zertifikat nach. Zusätzlich weist der Client aber auch mittels eigenem Zertifikat seine Identität gegenüber dem Server aus. Das Client-Zertifikat muss dazu durch eine *CA* ausgestellt sein, der der Server vertraut. Insbesondere *M2M*-Kommunikation kann mittels *mTLS* abgesichert werden, um eine Vertrauensstellung zwischen Geräten zu etablieren. Webbrowser und andere *UAs* können das Verfahren ebenfalls zur Authentifizierung nutzen. Möglichkeiten und Probleme von *mTLS* zur accountbasierten Nutzer-Authentifizierung werden in Kapitel 4.3.2 im Kontext der *MFA* behandelt. Auch ohne *mTLS* soll eine wechselseitige Authentifizierung durch die Applikation ermöglicht werden.

Für die Implementierung der Webapplikation wird der Einsatz von *TLS* im Fol-

[2] Empfehlungen zu konkreten Cipher-Suits inklusive zukünftigen Verwendungszeiträumen sind [BSI21] zu entnehmen.

genden als gegeben vorausgesetzt und nicht näher thematisiert. Das Referenz-Protokoll in Kapitel 2.3 wird unter Verwendung von *TLS* definiert, wobei die Perspektive der Applikation eingenommen wird: Beispielsweise bedeutet eine Klartext-Passwortübertragung eine mittels *TLS* geschützte Übertragung, bei der das Passwort für die Applikation auf Empfängerseite im Klartext vorliegt.

2.2.2 Föderierte Identitäten und Single-Sign-On

Typischerweise implementiert jede Applikation eine eigene Nutzerverwaltung und agiert damit als *IDP*. Entsprechend groß ist die Angriffsoberfläche, wenn für einen Nutzer an jeder betreffenden Applikation ein Account und damit eine separate Identität vorzuhalten ist. Föderierte Identitäten erlauben ein zentralisiertes Identity-Management mit nur einem *IDP* in einer Multi-Applikationsumgebung. Deshalb sollten Anwendungen föderierte Identitäten zugunsten oder ergänzend zur internen Nutzerverwaltung unterstützen. Dies gilt insbesondere für Applikationen, die in Unternehmen oder Institutionen eingesetzt werden.

Föderierte Identitäten werden klassischerweise als zentrale *DB* mit Nutzerstamm implementiert, die z. B. via *Lightweight Directory Access Protocol (LDAP)* in Applikationen eingebunden wird. Zur Implementierung von clientseitiger Passwortverarbeitung und *MFA* müssen Client- und Server-Komponente einer Webapplikation modifiziert werden. Die Integration dieser Features muss in jeder Applikation erfolgen, die accountbasiert arbeitet, auch wenn ein zentraler Nutzerstamm verwendet wird. Dieser Aufwand zur Etablierung der Features entfällt, wenn das Identity-Management aus der Applikation gelöst und als separates Framework betrachtet wird. Diese Abstraktion wird durch *Single Sign-On (SSO)*-Protokolle ermöglicht. Dabei überlässt die Dritt-Applikation, genannt *Relying Party (RP)*, dem *IDP* das Identity-Management vollständig und kommuniziert mit ihm über das *SSO*-Protokoll. Die Implementierung von Passwortverarbeitung und *MFA* beschränkt sich damit auf die *IDP*-Applikation. Diese Externalisierung setzt eine Vertrauensbeziehung der *RPs* zum zentralen *IDP* voraus.
Anzumerken ist, dass das Verfahren eine Gefahr beim Einsatz außerhalb von Institutionen mit bekanntem Personenkreis darstellt. Wenn Betreiber nutzerstarker Internet-Plattformen als *IDP* für unabhängige dritte Webseiten fungieren, können sie applikationsübergreifendes Nutzer-Tracking betreiben. Für die unabhängigen Dritten entsteht eine harte technische Abhängigkeit von besagten Plattformbetreibern, was dem Prinzip eines dezentralen Internets widerspricht.

Für *SSO* mittels föderierter Identitäten existieren Standard-Protokolle. Hierzu zählen Kerberos, *Security Assertion Markup Language (SAML)* (darauf aufbau-

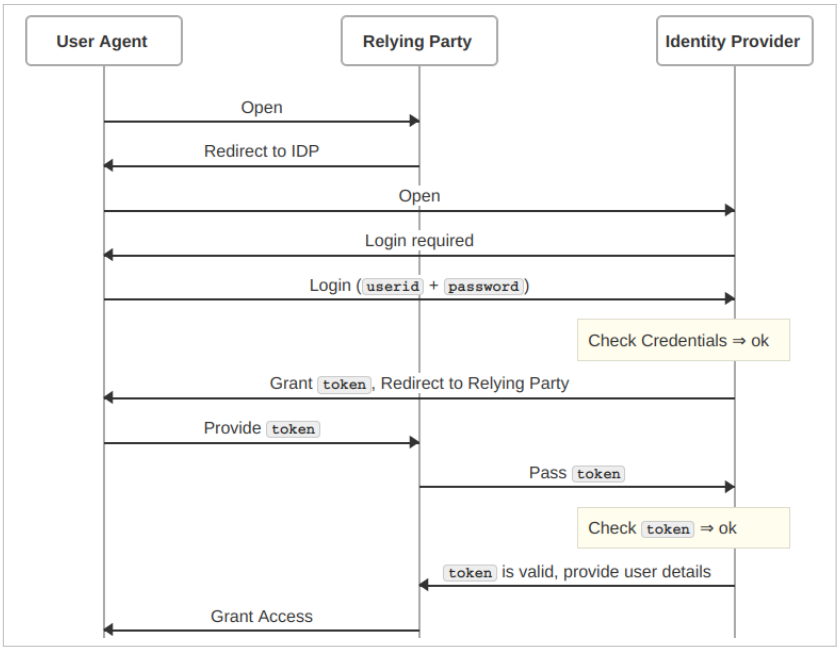


Abbildung 2.5: Exemplarischer Ablauf eines *SSO*-Logins inklusive Anmeldung am *IDP*
Quelle: Eigene Darstellung

end z. B. Shibboleth) und *OpenID Connect (OIDC)* auf Basis von OAuth2. Der Grundgedanke ist jeweils, dass der Nutzer seine Identität gegenüber dem *IDP* nachweist. Daraufhin stellt dieser einen Token (auch Code oder Ticket genannt) für den Nutzer aus. Möchte der Nutzer authentifizierten Zugriff zu einer *RP* erlangen, so muss er ihr den Token vorlegen. Die *RP* kann nun die Gültigkeit des Tokens sowie Informationen zum Nutzer beim *IDP* erfragen. Wenn der *IDP* den vorgelegten Token authentifiziert, so ist der Nutzer bei der *RP* authentisiert. Abbildung 2.5 zeigt den Ablauf exemplarisch.[Poh19, SPS11]

Durch ein zentrales *SSO* kann ein einheitliches Sicherheitsniveau bezüglich der Authentifizierung applikationsübergreifend etabliert werden. Ein *SSO*-fähiger *IDP* stellt damit die geeignete Grundlage zur Umsetzung von Verfahren der clientseitigen Passwortverarbeitung und optionalen *MFA* dar.

2.2.3 Hashing von Passwörtern

Passwort-Hashing stammt aus der Ära der Unix-Mainframes. Da bei diesen die Ablage der Credentials in der allgemein lesbaren Datei `/etc/passwd` erfolgt, ist eine Passwort-Speicherung im Klartext nicht möglich. Das Hashing schützt die Passwörter vor Einblicknahme durch andere Nutzer des Mainframes. Zur Authentifizierung gibt ein Nutzer seine Credentials im Terminal an. Das System prüft durch Hashing die Gleichheit mit dem zum Nutzer hinterlegten Hashwert und gestattet daraufhin den Login. [Con15]
Die Architektur verteilter Systeme, wie Webanwendungen, verhindert eine Ein-

blicknahme serverseitig gespeicherter Hashwerte durch Clients. Der Angriffsvektor besteht in Form irregulärer Zugriffe. Hierzu sind exemplarisch zu nennen:

- **Bugs**: Implementierungsfehler erlauben Angreifern die Exfiltration unter Umgehung der Begrenzungen durch die gebotenen Schnittstellen, z. B. via *SQL Injection (SQLi)*.
- ***Data Breaches***: Eine versehentliche Veröffentlichung führt zur unerlaubten Einblicknahme durch Dritte, z. B. durch ungeschützte Cloud-Backups.
- **Angreifer von Innen**: Die privilegierte Position des legitimen Zugriffs wird missbräuchlich genutzt, z. B. durch neugierige Administratoren.

Im Folgenden werden die Bedeutung der Geschwindigkeit sowie Eingabeparameter zum sicheren Hashing von Passwörtern erläutert.

Hashing-Aufwand Zu jedem Nutzer wird in der Datenbank ein Account mit Stammdaten gespeichert. Zu einem Account gehören die Credentials, bestehend aus Nutzerkennung uid und Passwort pw. Der Server bildet mittels kryptographischer Hashfunktion den Passwort-Hashwert: $pw' = hash(pw)$. Das Klartext-Passwort wird zu keinem Zeitpunkt persistiert. Sonst könnte ein Angreifer mit Zugang zu den Nutzerdaten direkt die Identität eines Opfers annehmen, indem er sich mit dessen Credentials anmeldet. Durch das *Hashing* wird der Aufwand der Account-Übernahme durch gestohlene Zugangsdaten um die Rückführung von pw aus pw' erhöht.
Kryptographische Hashfunktionen können in die Gruppen der schnellen und langsamen unterteilt werden. Eine schnelle Hashfunktion $hash^{fast}$ ist zum manipulationssicheren Vergleich großen Datenmengen konzipiert und damit auf eine hohe Datendurchsatzrate optimiert. Beispiele sind *Message-Digest Algorithm 5 (MD5)* und die *Secure Hash Algorithm (SHA)*-Familie. Ein zweckentfremdendes Passwort-Hashing mittels $pw' = hash^{fast}(pw)$ würde für einen Angreifer kaum eine Hürde darstellen, da er mit hoher Geschwindigkeit mögliche Passwörter ausprobieren kann. Stattdessen muss zum Passwort-Hashing eine langsame Hashfunktion verwendet werden: $pw' = hash^{slow}(pw, cost)$. Eine langsame Hashfunktion ist per Design als ineffizienter Algorithmus konzipiert. Aus der Ineffizienz ergibt sich die Sicherheit gegen Angriffe, sodass keine schnellere oder ressourcensparsamere als die bekannte Möglichkeit bestehen darf, aus einem Eingabedatum den zugehörigen Hashwert zu erzeugen. Eine Funktion $hash^{slow}$ erhält einen zusätzlichen Kostenparameter $cost$, über den der notwendige Ressourcenaufwand zur Erzeugung eines Hashwertes gesteuert wird. Die Prüfung zur Richtigkeit der Passwortangabe eines Nutzers bedarf nur einer Ausführung von $hash^{slow}$. Wäh-

renddessen muss ein Angreifer für jeden Passwort-Kandidaten $hash^{slow}$ ausführen. Dieses Ungleichgewicht macht Angriffe gegen das Hashing aufwändig und schlecht skalierbar. Hierin besteht die Sicherheitswirkung des Verfahrens.
Aus $hash^{fast}$ kann durch wiederholte Anwendung des Hashings eine Funktion $hash^{slow}$ abgeleitet werden, wobei *cost* dann der Anzahl an durchzuführenden Iterationen entspricht. Beispielsweise basiert die Funktion *Password-Based Key Derivation Function 2 (PBKDF2)* auf diesem Prinzip und nutzt intern ein *SHA*-Verfahren. Allerdings wird dadurch nur der Berechnungsaufwand bei moderatem Speicherverbrauch erhöht. Dies erlaubt den Einsatz von spezialisierten Mikrocontrollern mit hohem Berechnungsdurchsatz von Hashing-Operationen und geringer Speicheranbindung. Durch Erhöhung des Speicherbedarfs kann dieser Angriff mitigiert werden, z. B. durch den Einsatz der Algorithmen scrypt oder Argon2. Der Parameter *cost* setzt sich dann aus dem Bedarf an Rechenoperationen und Speicherbereichszugriffen zusammen.
Als $hash^{slow}$ kann eine dedizierte Hashfunktion, z. B. bcrypt, oder eine *Key Derivation Function (KDF)* verwendet werden. Eine *KDF* dient zur Ableitung von hochentropischem Schlüsselmaterial aus einem Eingabedatum, das niedrige oder hohe Entropie besitzen kann. Ein Passwort in Verwendung als Passwortkenntnis ist niedrigentropisch und das Hashing nur durch eine geeignete *KDF* vorzunehmen. Hierzu zählen u. a. *PBKDF2*, scrypt oder Argon2. Andere *KDF*, wie *HMAC-based KDF (HKDF)*, können nur aus hochentropischem Eingabematerial sichere Schlüssel ableiten. Letztere werden als *Key-Based KDF (KBKDF)* bezeichnet und sind nicht zum Passwort-Hashing geeignet.
In der Datenbank sollte ergänzend zu jedem pw' der verwendete Wert *cost* mit abgespeichert werden. Sollen die Kosten-Anforderungen erhöht werden, so kann der Hashwert mit dem neuen Kostenwert aktualisiert werden, sobald das Klartext-Passwort das nächste Mal am Server vorliegt, z. B. beim Login.
Der Lebenszyklus einer Software kann sich über Jahre oder Jahrzehnte erstrecken. Da mit der Zeit auch die Berechnungsfähigkeiten eines Angreifers potenziell steigen, sollten die Kosten-Anforderungen regelmäßig angepasst werden, z. B. jährlich. Außerdem sollte im Entwurf der Software die Möglichkeit vorgesehen werden, weitere Hashing-Funktionen zu unterstützen.
Wurde bereits ein Nutzerstamm aufgebaut und dazu eine Funktion $hash^{fast}$ zum Passwort-Hashing verwendet (oder der Kostenfaktor deutlich zu niedrig angesetzt), so kann eine einmalige Migration vorgenommen werden. Dazu wird jeder gespeicherten Hashwert pw'_{old} durch einen sicheren Wert pw'_{new} ersetzt:
$pw'_{new} := hash^{slow}(pw'_{old}, cost) = hash^{slow}(hash^{fast}(pw), cost)$
Meldet ein Nutzer sich das nächste Mal an, so ist der zu speichernde Hashwert zu aktualisieren: $pw' := hash^{slow}(pw, cost)$. Andernfalls bleibt das Verfahren nach

[Cro20] anfällig für Password-Shucking: Hat ein Angreifer aus Data-Breaches Tupel der Form $(uid, hash^{fast}(pw))$ erbeutet, so kann er versuchen, sie anhand der Nutzerkennung mit dem Nutzerstamm der vorliegenden Anwendung zu korrelieren. Der Suchraum kann bei erfolgreicher Verknüpfung und Wiederverwendung von Passwörtern durch Nutzer enorm eingeschränkt werden, sodass auch die Anwendung von $hash^{slow}$ keinen Schutz bietet. Das Password-Shucking ist an dieser Stelle als Spezialform des Credential-Stuffing anwendbar, siehe Kapitel 2.1.2.

Salt und Pepper Das Hashing mittels $hash^{slow}$ behindert Dictionary- und Brute-Force-Angriffe, da ihr Zeitbedarf erhöht wird. Brute-Force-Angriffe durchsuchen einen vordefinierten Suchraum möglicher Passwörter, indem der Hashwert von jedem Passwort-Kandidat gebildet wird, bis eine Übereinstimmung auftritt oder der Suchraum erschöpft ist. Dictionary-Attacken begrenzen den Suchraum auf den Inhalt eines Wörterbuches wahrscheinlicher Passwörter sowie typischer Variationen dieser Wörter.[Poh19] Diese Variationen werden durch Substitutionsregeln definiert, z. B. via Ableitung von *Leetspeak*-Formen der Wörter.
Allerdings bieten die bisherigen Maßnahmen keinen Schutz gegen vorberechnete Hashwerte für gängige Passwörter. Sei ein Angreifer gegeben, der über eine Lookup-Table in Form einer sortierten Liste vorberechneter Hashwerte und der Zuordnung der Original-Passwörter verfügt. Bei wahlfreiem Zugriff kann direkt ein Passwort ermittelt werden, wenn der zugehörige Hashwert in der Lookup-Table enthalten ist. Hat ein Angreifer mehrere Passwort-Hashes erbeutet, so kann er über einen Reverse-Lookup jedem betroffenen Nutzer ein Klartext-Passwort zuordnen, sofern der Hashwert in der Lookup-Table enthalten ist.[Hor19]
Eine derartige Lookup-Table für viele Hashwerte ist enorm speicherintensiv: Sei ein Hashingverfahren $hash$ mit $|hash| = 256$ gegeben, z. B. *SHA*-256. Sei Σ ein Alphabet mit $|\Sigma| = 62$ (Groß- und Kleinbuchstaben sowie Ziffern). Sei die Passwortmenge $\{pw \mid pw \in \Sigma^* \wedge 4 \leq |pw| < 8\}$ in der Lookup-Table abgebildet. Bei einem Encoding von einem Byte (8 Bit) je Zeichen ergibt sich ein Speicherbedarf in Bit von $\sum_{i=4}^{7}(62^i(256 + 8i)) \approx 127\ TiB$. Werden zusätzlich achtstellige Passwörter hinzugenommen, beträgt der Bedarf knapp 8 PiB. Durch Komprimierung der sortierten, fortlaufenden Hashwerte ließe sich der Speicherbedarf senken. Falls jedoch nur ausgewählte Wörter eines Wörterbuches enthalten sind, ist auch eine Komprimierung nicht mehr effektiv.
Der Ansatz bietet eine sehr geringe Zugriffszeit, aber skaliert aufgrund des enormen Speicherbedarfs nicht gut für größere Passwortmengen. Im Gegensatz dazu haben Brute-Force- und Dictionary-Angriffe einen geringen Speicherverbrauch aber eine sehr hohe Laufzeit. Einen ausbalancierteren *Time-Memory Tradeoff*

(TMTO) bieten *Rainbow-Tables*.

Eine *Rainbow-Table* ist eine Datenstruktur, die aus m Ketten besteht. Jede Kette $i = 0..m$ wird aufgebaut, indem ausgehend von einem gewählten Startpasswort $pw_{i,0}$ der Hashwert $pw'_{i,0} := hash(pw_{i,0})$ abgeleitet wird. Mithilfe einer Reduktionsfunktion *reduce* wird $pw_{i,1} := reduce(pw'_{i,0})$ bestimmt. Das Ziel der Reduktionsfunktion ist die Abbildung eines Hashwertes auf einen Passwort-Kandidaten. Die Kettenelemente $j = 0..n$ mit n als Kettenlänge werden abwechselnd als $pw'_{i,j} := hash(pw_{i,j})$ und $pw_{i,j+1} := reduce(pw'_{i,j})$ berechnet. Die Kettenbildung ist beendet, sobald der Endhashwert $pw'_{i,n}$ bestimmt wurde. Zu einer Kette i werden nur $pw_{i,0}$ und $pw'_{i,n}$ abgespeichert. Abbildung 2.6 veranschaulicht das Vorgehen. Der *TMTO* wird über die Kettenlänge gesteuert: Je länger diese sind, desto größer ist die Speicherersparnis, aber auch der Zeitbedarf zur Rekonstruktion.
Soll nun das Passwort zu einem gesuchten Hashwert spw'_0 gefunden werden, so wird jede Kette i auf $spw'_0 = pw'_{i,n}$ überprüft. Solange der Wert nicht gefunden wurde, sind in einer Schleife ausgehend von $k = 0..n-1$ die Werte $spw_{k+1} := reduce(spw'_k)$ und $spw'_{k+1} := hash(spw_{k+1})$ durch Inkrementieren von k zu bestimmen. Sobald $spw'_k = pw'_{p,n}$ für eine Kette p gilt, muss die Kette ausgehend von $pw_{p,0}$ rekonstruiert werden, um spw_0 zu finden. Falls die Schleife ohne einen Fund endet, ist das Passwort nicht in der *Rainbow-Table* enthalten.

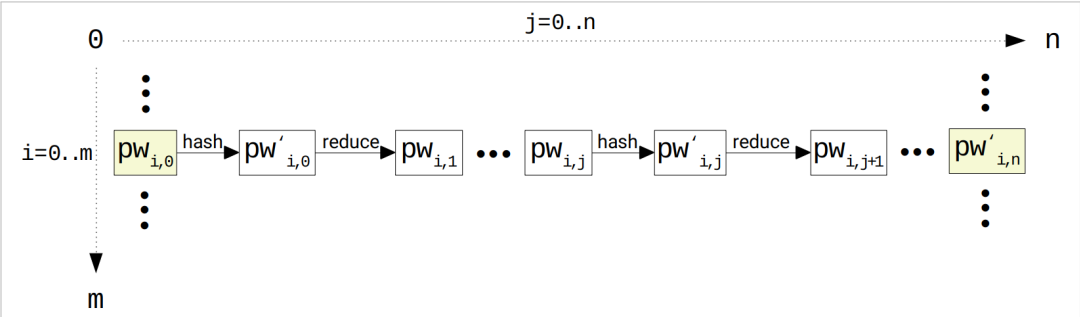


Abbildung 2.6: Schematisches Vorgehen zum Aufbau einer Rainbow-Table
Farblich hervorgehoben sind zu speichernde Werte. Quelle: Eigene Darstellung

Um das Passwort-Hashing vor diesen Angriffen mit vorberechneten Werten zu schützen, wird ein Passwort vor dem Hashing mit einem Zufallswert kombiniert, sodass auch gleichlautende Passwörter auf verschiedene Hashes abgebildet werden. Dadurch werden Funde in generischen *Rainbow-Tables* äußerst unwahrscheinlich und gezielte Dictionary-Attacks können zu Brute-Force-Angriffen degradiert werden. Das Verfahren wird als Salting bezeichnet. Ein Salt sollte durch einen *Cryptographically Secure Pseudo-Random Number Generator (CSPRNG)* gewählt und für jeden Account individuell sein.[Hor19] Als weitere Schutzmaßnahme wird zusätzlich zum Salt ein Pepper in das Passwort-Hashing einbezogen:

$pw' = hash^{slow}(salt \parallel pepper \parallel pw, cost)$ mit $\parallel$ als Konkatenationsoperator.
Ein Pepper erfüllt beim *Hashing* den gleichen Zweck wie das Salt, wird aber besonders geheim gehalten, z. B. indem es außerhalb der Datenbank abgelegt wird. Der Unterschied im Nutzen zwischen Salt und Pepper ist organisatorischer Natur: Erhält ein unberechtigter Dritter Zugriff auf die Datenbank, jedoch nicht auf den Speicherort des Peppers, so sind ihm nur die Salts, nicht aber das Pepper bekannt. Ein Pepper ist als zusätzliches Salt anzusehen, dessen Geheimhaltung die Angriffsresistenz der Hashes stärkt.[3] Mit Kenntnis von Salt und Pepper kann der Angreifer eine nutzerspezifische *Rainbow-Table* aufbauen. Diese lässt sich nicht für andere Accounts wiederverwenden, solange die Salts je Account individuell sind. Dazu sind die Salt-Suchräume hinreichend groß zu wählen.[Poh19] Beispielsweise ergeben sich aus einer Salt-Länge von vier Bytes $2^{32} \approx 4 * 10^9$ mögliche Salts.

Als abschließende Maßnahme sollte die Nutzerkennung ebenfalls in das Passwort-Hashing involviert werden, um Hashwerte an Accounts zu binden. Sei ein Angreifer mit eigenem Account an der Anwendung sowie lesendem und schreibendem Zugriff auf die Nutzer-*DB* gegeben. Die Maßnahme hindert den Angreifer daran, sich mit seinem Passwort an einem anderen Account anzumelden, indem er pw' und *salt* des Opfer-Accounts mit den Werten seines Accounts überschreibt. Die Maßnahme ist nur begrenzt nützlich: Ein externer Angreifer mit Schreibzugriff zur *DB*, z. B. via *SQLi*, könnte deutlich mehr Schaden als eine Impersonation-Attacke anrichten. Stattdessen wirkt die Maßnahme gegen interne Angreifer mit administrativem Datenbankzugang, die möglichst unbemerkt kurzzeitig Einblick in fremde Accounts nehmen möchten, z. B. im Unternehmenskontext zur Industriespionage.
Die Integration der Nutzerkennung in die Hashbildung impliziert, dass diese nicht ohne Passwortangabe änderbar ist. Die *uid* kann entweder direkt in das Hashing einbezogen oder auch zur Erzeugung nutzerspezifischer Pepper eingesetzt werden: Ausgehend vom statischen Secret *pepper* wird ein Nutzer-Pepper abgeleitet als $pepper'(uid) := kdf(pepper, uid)$. Die *uid* wird als Salt für die *KDF* eingesetzt. Diese Pepper-Konstruktion erlaubt das Herauslösen des *pepper* aus der Applikation. Damit ist es z. B. in ein separates Sicherheitsmodul integrierbar, das zu einer angefragten *uid* den Wert $pepper'(uid)$ zurückgibt, ohne *pepper* preiszugeben. Hierzu sind z. B. die *Trusted Platform Modules (TPMs)* der Server-Infrastruktur nutzbar, welche in Kapitel 4.3.3 aufgegriffen werden. Der Passwort-Hashwert

[3] In Anlehnung an die Parameter *salt* und *pepper* wird *cost* auch als Garlic bezeichnet.

eines Accounts ergibt sich abschließend als:

$$pw' := hash^{slow}(salt \parallel kdf(pepper, uid) \parallel pw, cost)$$

Wie eingangs aufgeführt, schützt Passwort-Hashing nur vor irregulären Zugriffsversuchen, wie *SQLi* oder *Data Breaches*. Ein Schutz regulärer Anmeldeversuche, z. B. an einem Login-Formular, ist damit nicht gegeben, auch wenn diese Anmeldeversuche aus einer Brute-Force-Attacke resultieren könnten. Hiergegen wird ein Rate-Limiting von Anfragen benötigt, siehe Kapitel 5.1.5.

2.2.4 Tokenbasierte Authentifizierung

Eine Authentifizierung mit den Credentials *uid* und *pw* erlaubt die Tätigung sämtlicher Operationen in der Applikation und stellt damit die höchste Berechtigungsstufe dar. Jedoch benötigt nicht jede Aktion die höchsten Berechtigungen. Über tokenbasierte Authentifizierung kann das *Principle of Least Privilege (PoLP)* umgesetzt werden. So ist ein Token bezüglich Einsatzzweck, Einsatzzeitraum sowie weiterer, kontextspezifischer Constraints einschränkbar. Der Server stellt Tokens aus und übergibt sie an die anfragenden Clients. Ein Client veranlasst die Ausführung einer geschützen Server-Aktion, indem er den passenden Token am Server vorlegt und dieser ihn erfolgreich validiert. Da der Client als Token-Träger fungiert, werden sie Bearer-Tokens genannt.[JH12]
Bearer-Tokens können als clientgestützt oder servergestützt klassifiziert werden. Die Einteilung erfolgt aufgrund des Speicherortes der Informationen zum Token. Beide Varianten werden im Weiteren benötigt und nachfolgend erläutert.

Server-gestützte Bearer-Tokens

Mittels servergestützter Bearer-Tokens werden klassischerweise Sitzungsverwaltungen implementiert. Das Konzept des Referenz-Protokolls hierzu wird in Kapitel 2.3.3 erläutert. Wenn für einen Nutzer mit der Kennung *uid* ein Token auszustellen ist, so erzeugt der Server zunächst mittels *CSPRNG* ein *Opaque*. In der *DB* wird ein Record mindestens bestehend aus $(uid, opaque, t_0)$ mit t_0 als Ausstellungszeitpunkt abgespeichert. Neben der Nutzerreferenz können weitere Informationen zum Token in der *DB* hinterlegt werden. Der Bearer-Token entspricht dem *Opaque*. Er wird dem Client mitgeteilt. Der Client legt den Token bei Requests vor, die eine tokenbasierte Authentifizierung erfordern. Wird der Token z. B. als Sitzungsmerkmal verwendet, so weist der Client hierüber seine Sitzung nach. Da potenziell bei jeder Anfrage die Token-Informationen aus der *DB* zu

beziehen sind, ist ein schneller Lookup des *Opaques* obligatorisch. Von einer Indizierung der *Opaques* ist aufgrund ihrer Länge und schlechten Sortierbarkeit abzusehen. Ein ressourceneffizienterer Ansatz besteht in der Aufnahme eines indizierten Identifiers in die Token-Konstruktion, welcher den schnellen Lookup ermöglicht. Hierzu könnte eine separate Token-ID als künstliches Attribut eingeführt werden. Nachfolgend wird stattdessen die bereits vorhandene und indizierte *uid* eingesetzt. Der Token ergibt sich als $serial((uid, opaque))$ mit *serial* als Funktion zur Serialisierung des Tupels.
Ein Token muss invalidierbar sein, z. B. um den Logout einer Session zu ermöglichen. Ansonsten könnten Token-Records unbegrenzt in der *DB* verbleiben. Zur Invalidierung servergestützter Tokens genügt das Löschen des Records zum *opaque* aus der *DB*. Der Client kann den *Opaque* daraufhin nicht mehr einsetzen. Die Invalidierung kann durch Client oder Server initiiert werden. Sobald die zeitlich beschränkte Lifetime des Tokens überschritten ist, invalidiert ihn der Server. Dazu gilt es die *DB* regelmäßig auf abgelaufene Tokens zu überprüfen, um diese zu löschen. Hierzu werden der zu jedem Token hinterlegte Wert t_0 sowie eine durch die Applikation festgelegte Token-Lifetime Δt verwendet. Ein abgelaufener Token bleibt effektiv gültig, solange er noch nicht gelöscht wurde. Als Gegenmaßnahme ist zusätzlich bei jedem *DB*-Abruf des Tokens die Einhaltung des Gültigkeitszeitraums zu prüfen.
Vorteilhaft am Verfahren ist die geringe Komplexität. Nachteilig sind die Speicherzugriffe auf die *DB* für jeden Token-Abruf, was die Skalierbarkeit beschränkt.

Client-gestützte Bearer-Tokens

Client-gestützte Bearer-Tokens besitzen eine höhere Komplexität, funktionieren dafür aber ohne Datenspeicherung im Backend. Anstatt eines *Opaques* enthält der Bearer-Token selbst die zugehörigen Informationen. Von diesen, im Weiteren Claims genannt, sind für die Betrachtung folgende relevant:

- $\mathbf{t_0}$: Ausstellungszeitpunkt des Tokens (Beginn des Gültigkeitszeitraums)
- $\mathbf{t_1}$: Ablaufzeitpunkt des Token (Ende des Gültigkeitszeitraums)
- **uid**: Kennung des Accounts, zu dem der Token ausgestellt wurde
- **type**: Verwendungszweck (zur Unterscheidung von Token-Typen)
- **tid**: Möglichst einmalige ID (*Nonce*) zur Token-Identifikation
- Zusätzliche, kontextabhängige Angaben zum Token (weitere Nutzlast)

Die Nutzlast ergibt sich als $body := serial((t_0, t_1, uid, tid, type))$. Der Server nutzt eine *Message Authentication Code (MAC)*-Funktion zur Signatur dieser Werte, wobei im Folgenden *Keyed-Hash MAC (HMAC)* verwendet wird. Der geheime

Wert *secret* ist der Schlüssel zur Generierung und Verifikation der Signatur. Es handelt sich um ein symmetrisches Signaturverfahren, da sowohl signierende als auch verifizierende Partei den gleichen Schlüssel kennen müssen. Beide Parteien werden durch den Server ausgefüllt, welcher *secret* geheimhalten muss. Im Gegensatz dazu erlauben asymmetrische Signaturverfahren getrennte Parteien für Signatur und Verifikation, siehe Kapitel 2.2.1 sowie 4.3. Symmetrische Signaturverfahren sind aufgrund der besseren Performance wenn möglich vorzuziehen. Angriffe auf sie werden als Teil von Kapitel 3.2.1 betrachtet.
Zur Ausstellung eines Bearer-Tokens bildet der Server Nutzlast, Signatur sowie den zu versendenden Token selbst als:

$$
\begin{aligned}
&\mathit{body} \coloneqq \mathit{serial}((t_0, t_1, \mathit{uid}, \mathit{tid}, \mathit{type})) \\
&\mathit{sign} \coloneqq \mathit{hmac}(\mathit{body}, \mathit{secret}) \\
&\mathit{token} \coloneqq \mathit{serial}((\mathit{body}, \mathit{sign}))
\end{aligned}
$$

Legt ein Client einen Token token_{inp} am Server vor, so muss der Server ihn zunächst verifizieren. Dazu teilt er ihn in seine Bestandteile auf, bildet die Signatur und prüft auf Gleichheit:

$$
\begin{aligned}
&(\mathit{body}_{inp}, \mathit{sign}_{inp}) \coloneqq \mathit{serial}^{-1}(\mathit{token}_{inp}) \\
&\mathit{sign}_{chk} \coloneqq \mathit{hmac}(\mathit{body}_{inp}, \mathit{secret}) \\
&\mathit{sign}_{inp} = \mathit{sign}_{chk}
\end{aligned}
$$

Zusätzlich zu prüfen sind die Einhaltung des Gültigkeitszeitraums (t_0, t_1) sowie die Übereinstimmung des Token-Typs für die angeforderte Aktion. Die Zeitstempel müssen zeitzonen-unabhängig definiert sein, z. B. via *Epoch*. Sonst könnten auch abgelaufene Tokens akzeptiert werden, wenn das Server-Deployment sich über mehrere Zeitzonen erstreckt. Außerdem muss die Systemzeit auf allen token-verarbeitenden Serverinstanzen synchron gehalten werden. Weil Bearer-Tokens lediglich signiert, die Nutzlast *body* aber nicht verschlüsselt ist, dürfen sie keine sensitiven Daten enthalten.

Da anstatt des *DB*-Speichers nur das statische *secret* über alle Server-Instanzen verteilt wird, ist der Ansatz sehr effektiv skalierbar. Nachteilig ist der Berechnungsaufwand für die *HMAC*-Signaturen. Damit dieser nicht bei jedem Request anfällt, können validierte Tokens in einem Cache vorgehalten werden. Davon profitieren Tokens, die häufig am Server vorgelegt werden, z. B. Session-Tokens. Hierzu werden *Least Recently Used (LRU)*-Caches eingesetzt, welche die letzten n vorgelegten Tokens mit gültiger Signatur aufnehmen.

Zur Referenzierung des Accounts und damit Authentifizierung des Nutzers dient der Claim *uid* im Token. Denkbar wäre auch eine tokenbasierte Autorisierung durch Aufnahme von Berechtigungen in den Bearer-Token. Dies führt jedoch zu einer *Privilege Escalation*, wenn einem Nutzer mit gültigem Token Rechte oder Account-Zugang zu entziehen sind. Da der Token noch auf die ursprünglichen Rechte verweist, würden sie dem Nutzer weiterhin zugestanden werden. Eine Gegenmaßnahme wäre die Invalidierung von Tokens, die vor dem Zeitpunkt der letzten Änderung der Nutzer-Berechtigungen ausgestellt wurden. Allerdings müsste dieser Zeitpunkt dann wiederum in der *DB* abgespeichert und bei jedem Request überprüft werden. Stattdessen könnten auch direkt die Nutzerberechtigungen aus der *DB* erfragt werden. Um von clientgestützten Tokens profitieren zu können, ist folglich von einer Realisierung der Autorisierung durch sie abzusehen. Alternativ wäre für unkritische Anwendungen das Risiko einer durch die Token-Lifetime zeitlich beschränkten Privilege Escalation bewusst zu akzeptieren.

Der Grund für die Untauglichkeit zur Abbildung von Autorisierungen liegt im transaktionalen Charakter clientgestützter Tokens begründet.[Rao21] Die Nutzerberechtigungen existieren außerhalb des Kontextes der Transaktion eines Tokens und sind damit nicht zugreifbar. Deshalb sind diese Tokens ebenfalls ungeeignet zur Umsetzung von Logouts: Möchte der Nutzer den Token vor Ende des Gültigkeitszeitraums invalidieren, so kann dies nicht über den Token abgebildet werden, da er weiterhin gültig und verwendbar bleibt. Folglich ist der Token über die Umgebung, in der er verwendet wird, zu invalidieren. Eine primitive Lösung ist das Löschen des Tokens vom Client, sodass dieser nicht mehr bekannt ist. Der Token ist damit jedoch nicht invalidiert und könnte durch einen Implementierungsfehler weiterhin im Client vorhanden sein, z. B. als Teil eines Backups oder Caches.
Stattdessen kann auch das zur Token-Signatur verwendete *HMAC*-Secret geändert werden, sodass eine Verifikation fehlschlägt. Dies ist jedoch nur in bestimmten Use-Cases möglich, siehe Kapitel 2.3.4. Deshalb wird zur Invalidierung clientgestützter Bearer-Tokens eine *Token Revocation List (TRL)* genutzt. Diese Liste wird im Backend vorgehalten und enthält die ausgeloggten Tokens, deren Lifetime noch nicht abgelaufen ist. Für einen vorgelegten Token hat der Server zusätzlich sein Fehlen auf der *TRL* zu überprüfen, um ihn anzuerkennen. Um den Speicherbedarf zu senken, wird anstatt eines Tokens nur die enthaltene Nonce *tid* auf der *TRL* vermerkt. Es gilt die Token-Lifetime so zu wählen, dass das erwartbare Token-Aufkommen der *TRL* möglichst gering ist. Dies wird in Kapitel 2.3.3 weiter ausgeführt.

In [JBS15] wird mit *JSON Web Tokens (JWTs)* ein bekannter Standard für clientgestützte Bearer-Tokens definiert. Zur Strukturierung der Nutzlast dient das

Datenformat *JavaScript Object Notation (JSON)*. Einige Token-Claims werden vorgegeben, um eine Interoperabilität zwischen Systemen zu erzielen. Zu diesen zählen die hier vorgestellten Claims.

2.3 Referenz-Protokoll

Bevor clientseitige Passwortverarbeitung (Kapitel 3) und optionale *MFA* (Kapitel 4) evaluiert werden, soll zunächst ein Referenz-Protokoll für den *IDP* ohne diese Merkmale definiert werden. Gegenüber dieser Referenz gilt es Sicherheitsverbesserungen zu erzielen und erläuterte Anforderungen umzusetzen. Das Referenz-Protokoll orientiert sich am Stand der Technik und einer Architektur mit minimaler Komplexität. Zur *MITM*-Prävention ist bei den nachfolgend skizzierten Verfahren ein sicherer Kommunikationskanal obligatorisch, siehe Kapitel 2.2.1. In den folgenden Abschnitten werden die durch das *IDP*-Protokoll zu realisierenden Funktionen erläutert, welche in Kapitel 2.1.1 eingeführt wurden.

2.3.1 Registrierung und Verifikation neuer Benutzer

Zur Registrierung ruft der Nutzer die Applikation auf und gibt seine Stammdaten an, die an den Server übermittelt werden. Von diesen sind nachfolgend die Nutzerkennung *uid*, die E-Mail-Adresse *mail* sowie das Passwort *pw* relevant. Der Server legt daraufhin einen neuen Account an, welcher zusätzlich durch den Nutzer zu verifizieren ist. Nachfolgend werden Formen der Registratur, Passwort-Hashing und Account-Verifikation sowie Mehrfach-Registrierungen betrachtet.

Formen der Registratur

Prinzipiell können offene und geschlossene Registratur unterschieden werden. Bei einer offenen legt der Nutzer den Account inklusive Stammdaten selbstständig an. Bei einer geschlossenen wird er zugewiesen, z. B. durch eine Personalabteilung. Jeder Account besitzt mit der *uid* einen eindeutigen Identifier. Wenn der Nutzer bei Registrierung seine *uid* frei wählen kann, muss ihm mitgeteilt werden, ob die gewünschte *uid* noch verfügbar ist. Damit ergibt sich das Problem einer User-Enumeration-Attacke, da ein Angreifer nun trivial feststellen kann, ob ein Account existiert.
Die Organisation *Open Web Application Security Project (OWASP)* empfiehlt in ihren Security-Guidelines eine User-Enumeration zum Schutz der Nutzer zu verhindern, indem generische Fehlermeldungen verwendet werden.[OWA21] Jedoch könnte ein Angreifer im geschilderten Szenario trotzdem auf Verdacht Accounts anlegen und vermerken, für welche Nutzerkennungen dies fehlschlägt. Damit wä-

re kein Sicherheitsgewinn bei verminderter *User Experience (UX)* gegeben. Die *OWASP*-Empfehlung funktioniert daher nur in folgenden Fällen:

- Eine geschlossene Registratur (ohne Wahlmöglichkeit der *uid*)
- Zuweisung einer generierten *uid* als technischer Schlüssel (kann die Personalisierung des Accounts senken und damit der *UX* schaden)
- Verwendung einer nutzerbezogenen ID mit Eindeutigkeit auch außerhalb des Applikationskontextes, wie Telefonnummer oder E-Mail-Adresse

Die drei Varianten lösen mögliche Kollisionen auf, ohne sie an den Nutzer zu propagieren. Variante drei setzt voraus, dass die externe Entität dauerhaft einem Nutzer zuzuordnen ist, jedoch können z. B. Telefonnummern den Besitzer wechseln. Weiterhin sind diese externen Merkmale in der Regel geheim zu halten, z. B. die Mail-Adresse zur Vermeidung von Spam. Damit sind sie nicht in der Applikation verwendbar, wenn nutzerbezogene Inhalte für andere bereitgestellt oder geteilt werden sollen. Deshalb werden *uid* und *mail* separat behandelt. Das Design des Referenz-Protokolls soll die Informationspreisgabe durch User-Enumerations verhindern. Die Referenz-Implementierung sieht jedoch eine Wahlfreiheit der *uid* durch den Nutzer vor. Deshalb wird die User-Enumeration auf einen *API*-Endpunkt des Servers beschränkt. Dieser Endpunkt ist durch Rate Limiting zusätzlich zu schützen, siehe Kapitel 5.1.5.

Passwort-Hashing

Zur Registrierung werden *uid*, *mail* und *pw* an den Server übertragen, welcher den Hashwert nach dem Verfahren aus Kapitel 2.2.3 bildet:

$$pw' := hash^{slow}(salt \parallel kdf(pepper, uid) \parallel pw, cost)$$

In der *DB* wird ein Account-Record bestehend aus $(uid, mail, salt, pw', cost)$ abgelegt. Der *pepper* ist als statisches Geheimnis gesondert außerhalb der *DB* zu speichern und besonders zu schützen, siehe Kapitel 2.2.3.

Account-Verifikation

Die Verifikation eines neu registrierten Accounts verlangt einen Nachweis über den Besitz eines Identitätsmerkmals bei einer externen Entität, was nachfolgend als Hinterkanal bezeichnet wird. Hierbei kann es sich z. B. um eine E-Mail-Adresse, Telefonnummer oder auch Postanschrift handeln. Der Hinterkanal dient u. a. als Vertrauensanker zum Zurücksetzen der Anmeldedaten, siehe Kapitel 2.3.4. Nachfolgend wird exemplarisch der E-Mail-Versand als Hinterkanal mit der angegeben

Mail-Adresse *mail* verwendet. Der Nachweis wird mittels Challenge-Response-Verfahren erbracht, welche Kapitel 3.3 vertieft. Die Challenge wird in Form eines Bearer-Tokens an das Mail-Konto versendet. Der Nutzer legt den Token als Response beim Server vor, um den Besitz des Mail-Kontos nachzuweisen. Damit ist der Account verifiziert und ein Login an der Applikation wird gestattet.
Die Konzepte zu Bearer-Tokens aus Kapitel 2.2.4 sind für den Prozess geeignet. Falls davon auszugehen ist, dass der Nutzer den Token manuell eingeben muss, sollte er aufgrund der Längenbegrenzung servergestützt sein. Dies gilt insbesondere beim Empfang auf einem separaten Gerät, z. B. als *Short Message Service (SMS)*-Nachricht. Für den Hinterkanal Mail wird diese Einschränkung nicht angenommen, sodass der Token clientgestützt sein kann.

Die Modellierung der Mail-Verifikation mittels clientgestütztem Token hat das Potenzial, eine *DB*-Speicherung zu vermeiden bis der Account verifiziert ist. Dazu enthält der Token dann sämtliche Stammdaten des Nutzers. Nach Verifikation wird mit den Daten aus dem Token der Account in der *DB* angelegt. Dieser Ansatz skaliert jedoch nur begrenzt: Werden bei Registrierung große Datenmengen erfasst (z. B. Profilbilder), so müssen diese vollständig in den Token aufgenommen werden. Eine Bereitstellung des Tokens in der Verifikationsmail als anklickbare *Uniform Resource Locator (URL)* oder *Quick-Response (QR)*-Code ist dann mitunter nicht möglich. Beide Formate können nur begrenzte Datenmengen transportieren.[4] Eine weniger integrative Bereitstellungsform könnte der *UX* schaden. Stattdessen wird auch für nicht-verifizierte Accounts ein *DB*-Record gespeichert, welcher um ein Flag $verified \in \{0, 1\}$ zum Status der Verifikation ergänzt wird. Zur Speicherbegrenzung sollten nicht-verifizierte Accounts eine begrenzte Lifetime $\Delta t_{verified=0}$ besitzen, nach deren Ablauf ein Cleanup-Job sie aus der *DB* löscht. Dazu werden Account-Records um den Zeitstempel der letzten Accountdaten-Änderung t_{change} ergänzt. Die Nutzlast der client-gestützten Bearer-Tokens umfasst damit mindestens folgende Angaben:

- Gültigkeitszeitraum: $[t_{change}; t_{change} + \Delta t_{verified=0}]$
- Stammdaten (*uid*, *mail*)
- Token-Typ: „account-verification“

Abbildung 2.7 fasst den bisher erläuterten Ablauf zusammen: Der Client fordert eine Registrierung an. Der Server bildet den Passwort-Hash und speichert den nicht-verifizierten Account in der *DB*. Anschließend generiert er eine Verifikations-*URL*, die den Token enthält und per Mail versendet wird. Der Nutzer ruft die *URL* über den *UA* auf. Der Server prüft den vorgelegten Token und markiert den

[4] Eine *URL* nimmt implementierungsabhängig in der Regel einige tausend Zeichen auf, typischerweise 2 000 bis 8 000. Diese Größenordnung gilt auch für *QR*-Codes.

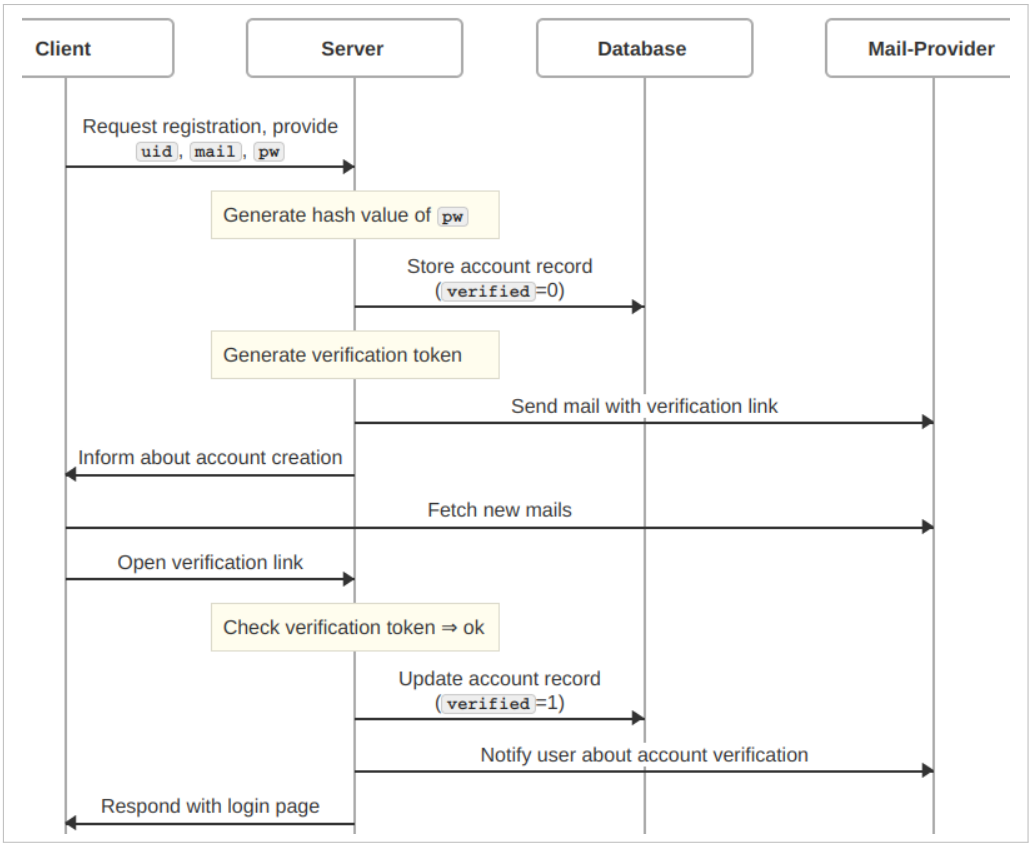


Abbildung 2.7: Referenz-Protokoll zur Registrierung ohne Sonderfallbehandlung
Quelle: Eigene Darstellung

Account als verifiziert. Die Verifikation des Accounts invalidiert den Token.

Mehrfach-Registrierungen

Bisher wurden keine Sonderfälle berücksichtigt. Wird eine Registrierung für eine vergebene *uid* angefordert, ist die Anfrage durch den Server abzuweisen. Wird zur Registrierung eine bereits verifizierte *mail* angegeben, so muss eine Erfolgsmeldung zurückgegeben werden, um die Mail-Adresse geheim zu halten. Der Mail-Kontoinhaber kann optional über den Registrierungsversuch informiert werden, was zur Spam-Vermeidung zu limitieren ist. Andernfalls könnte ein Angreifer Registrierungsanfragen zu einer bekannten Mail-Adresse versenden, bis der Spam-Filter die Mails der Anwendung als Spam klassifiziert. Der Nutzer würde dann legitime Mails der Applikation nicht mehr empfangen. Also z. B. auch solche zur Zurücksetzung der Anmeldedaten, wie in Kapitel 2.3.4 beschrieben.
Wird eine weitere Registrierung für einen nicht-verifizierten Account angefordert, so muss die neue Registrierungsanfrage den Record in der *DB* überschreiben. Andernfalls könnte ein Spammer viele Nutzerkennungen blockieren, indem er sie regelmäßig registriert und nie verifiziert. Der Server muss bei Verifikation zusätzlich prüfen, dass die Mail-Angaben in Token und *DB* übereinstimmen. Andern-

falls könnte der veralteten Token zu einer überschriebenen Registrierungsanfrage trotzdem verwendet werden. Wenn der Nutzer den Account nicht innerhalb von $\Delta t_{verified=0}$ verifiziert, muss er eine neue Registrierungsanfrage stellen.

```
1  if uid ∈ Database ∧ verified(uid)=True:
2      error: uid already taken
3  else:
4      (salt, cost, pw′) = generate_user_password_hash(pw, uid, pepper)
5      if uid ∈ Database:
6          update_account_record_by_uid(uid, mail, salt, cost, pw′)
7      else:
8          create_account_record(uid, mail, salt, cost, pw′)
9      token = generate_verification_token(uid, mail)
10     if mail ∈ Database ∧ verified(mail)=True:
11         send_info_mail(mail)  # notify mail-addr owner about registration attempt
12     else:
13         send_verification_mail(mail, token)
```

Listing 2.1: Bearbeitung einer Registrierungsanfrage durch den Server

Listing 2.1 fasst die Server-Schritte als Pseudocode zusammen. Der Code ist nicht auf Performance, sondern auf Ausführung in möglichst konstanter Zeit optimiert. Deshalb werden Operationen mitunter auch ohne Bedarf dazu ausgeführt. Dies soll verhindern, dass ein Angreifer eine kürzere Bearbeitungsdauer als Seitenkanal nutzt, z. B. zur Feststellung ob eine Mail-Adresse als verifiziert hinterlegt ist. Dazu darf ab Zeile 4 die Bearbeitungsdauer der Aktionen in den Verzweigungen nicht signifikant voneinander abweichen.
Vom alternativen Ansatz, die Bearbeitungsdauer durch Einfügen zufälliger Wartezeiten zu verschleiern, wird abgeraten: Die Wartezeiten müssten sich stufenlos aus einem Intervall mit definiertem Minimum sowie Maximum ergeben. Die Intervallgröße müsste sich an erwartbaren Ausführungszeiten orientieren, welche jedoch abhängig vom Deployment sind. Ein Angreifer könnte mit hinreichend vielen Requests aufgrund der zufälligen Wartezeit-Verteilung dennoch eine signifikante Abweichung feststellen. Dagegen müssten adaptive Wartezeiten über Messungen zur Ausführungszeit der Operationen etabliert werden. Die Komplexität wäre erhöht und das Verfahren fragil gegenüber starken Lastschwankungen, welche wiederum gezielt durch einen Angreifer herbeigeführt werden könnten.

2.3.2 Anmeldung registrierter Nutzer

Zur Anmeldung legt der Nutzer *uid*, oder alternativ *mail*, sowie *pw* am Server vor. Dieser bezieht den Account-Record aus der *DB* und überprüft, dass der Account verifiziert wurde. Daraufhin bildet er den Passwort-Hashwert, siehe Kapitel 2.2.3. Bei Übereinstimmung mit dem hinterlegten Hashwert ist der Nutzer

authentifiziert. Anschließend wird überprüft, ob der Wert *cost* des Accounts den aktuellen Anforderungen entspricht, siehe Kapitel 2.2.3. Falls nicht wird *cost* aktualisiert sowie pw' neu gebildet und beide Werte werden in die *DB* geschrieben. Zum Abschluss der Anmeldung wird eine Sitzung erzeugt und zurückgegeben, siehe Kapitel 2.3.3.

```
1  record_account := query_verified_account_by_uid_or_mail(uid_request, mail_request)
2  if record_account ∈ Database:
3      (uid_db, pw'_db, salt_db, cost_db) := record_account
4      pw'_request := hash_password(pw_request, uid_db, salt_db, pepper, cost_db)
5      ok := (pw'_db = pw'_request)
6  else:
7      (uid_fake, pw'_fake, salt_fake, cost_fake) := record_fake
8      pw'_request := hash_password(pw_request, uid_fake, salt_fake, pepper, cost_fake)
9      ok := ( pw'_fake = pw'_request) ∧ False )
10 if ok:
11     if cost_db ≠ cost_wanted:
12         pw'_update := hash_password(pw_request, uid_db, salt_db, pepper, cost_wanted)
13         update_account_record(pw'_update, cost_wanted)
14     create_session(uid)
15 else:
16     error: uid or mail or pw wrong
```

Listing 2.2: Anweisungssequenz zur Anmeldung mit Fehlerbehandlung

Listing 2.2 zeigt den Prozess als Pseudocode. Da sowohl *uid* als auch *mail* als Merkmale zum Login zugelassen sind, kann eine einfache Konfusionsattacke ausgeführt werden: Seien zwei verifizierte User durch $(uid_1, mail_1)$ und $(uid_2, mail_2)$ mit $uid_2 = mail_1$ gegeben. Implementierungsabhängig kann sich nun der erste Nutzer nicht mehr mit $mail_1$ oder der zweite nicht mehr mit uid_2 anmelden. Deshalb muss die Schnittmenge von Nutzerkennungen und Mail-Adressen leer sein. Hierzu kann beispielsweise das „@“-Zeichen in Mail-Adressen vorausgesetzt und in Nutzerkennungen verboten werden.
Erst ab Zeile 10 von Listing 2.2 ist der Nutzer authentifiziert. Davor dürfen die Verarbeitungszeiten fallunabhängig nicht signifikant voneinander abweichen. Andernfalls stellt sie einen Seitenkanal dar, über den z. B. eine User-Enumeration durchführbar wäre. Deshalb wird für nicht-existente oder nicht-verifizierte Accounts ein Fake-Record verwendet. Die Werte des Fake-Records werden bei Programmstart zufällig erzeugt.

Für die Hashvergleiche (Zeile 5 und 9) ist eine zeitsichere Vergleichsfunktion zu nutzen. Reguläre Vergleichsfunktionen brechen die Operation beim ersten abweichenden Bit ab. Dadurch wird potenziell eine Zeitmessung als Seitenkanal ermöglicht. Diese könnte als Orakel verwendet werden, um die Übereinstimmung der Anfangsbits beider Hashwerte sukzessive zu erhöhen. Der Zeitaufwand würde von der Größenordnung $\mathcal{O}(2^n)$ auf $\mathcal{O}(n)$ reduziert werden. Eine zeitsichere

Vergleichsfunktion verhindert dies, indem das Laufzeitverhalten zweier Eingaben mit bitgleicher Länge wertunabhängig gehalten wird. Diese Anforderung wird an sämtliche Vergleiche von Geheimniswerten gestellt. Anzumerken ist, dass das Gelingen eines solchen Angriffs unwahrscheinlich ist, wenn mehrere Hops zwischen Client und Server liegen, wie im Internet üblich.[Hor19] Das Protokoll sollte jedoch nicht auf dieser Annahme zur Umsetzung aufbauen.

Beim Scheitern einer Anmeldung darf zum Schutz vor User-Enumerations der Fehlergrund nicht kommuniziert werden, siehe Kapitel 2.3.1. Stattdessen sollte der Server eine generische Fehlermeldung mit möglichen Gründen zurückgeben.

2.3.3 Sitzungsverwaltung angemeldeter Nutzer

Das Sitzungskonzept erweitert den Ansatz tokenbasierter Authentifizierung aus Kapitel 2.2.4. Bei jedem Request im Kontext der Sitzung muss der Client dem Server seinen Token übermitteln. Ein Logout entspricht der Token-Invalidierung. Die Sitzungsverwaltung wäre vollständig mit servergestützten Tokens umsetzbar, was jedoch mit den erläuterten Nachteilen von serverseitigem Zustand einhergeht. Deshalb wird stattdessen eine Kombination von client- und servergestützten Tokens eingesetzt.

Nach Anmeldung wird die Session in Form eines Token-Paares $(token_{acc}, token_{ref})$ erstellt. Der Access-Token $token_{acc}$ ist clientgestützt und hat eine kurze Lebensdauer, z. B. wenige Minuten. Er erlaubt den Zugriff auf die Anwendung. Der Refresh-Token $token_{ref}$ ist servergestützt und hat eine längere Lebensdauer, z. B. Tage oder Monate. Er erlaubt in Kombination mit einem abgelaufenen das Ausstellen eines neuen Access-Tokens. Der Einsatz der Access-Tokens senkt die Anfragelast auf die Datenbank und erhöht die Skalierbarkeit des Backends. Durch Einsatz der Refresh-Tokens werden die Probleme der Revocation client-gestützter Tokens verringert, da kurze Lebenszeiten für die Access-Tokens wählbar sind.

Bei jeder Erneuerung eines Access-Tokens sollte der *Opaque* des zugehörigen Refresh-Tokens geändert werden, damit ein Angreifer keine ihm bekannten, älteren Tokens erneuern kann. Um Access- und Refresh-Token aneinander zu koppeln, wird die Token-ID *tid* genutzt, siehe Kapitel 2.2.4. Diese wird zum Refresh-Token hinterlegt, sodass die Session an die *tid* gekoppelt ist. Zum Logout reicht

dann durch Angabe der *tid* einer der beiden Tokens aus. Damit ergibt sich:

$$body_{acc} := serial((t_{acc_0}, t_{acc_1}, uid, tid, \text{`}access\text{`}))$$
$$token_{acc} := serial((body_{acc}, hmac(body_{acc}, secret)))$$
$$token_{ref} := serial((tid, opaque))$$

Zu einer Session ist der *DB*-Record $(tid, uid, t_{ref_0}, opaque)$ mit t_{ref_0} als Erstellungszeitpunkt zu hinterlegen. Für Refresh-Tokens muss eine begrenzte Lifetime Δt_{ref} definiert sein. Abbildung 2.8 fasst die erläuterten Abläufe zusammen.

Da die Sessions in der *DB* gespeichert sind, kann einem Nutzer im Frontend eine Auflistung seiner aktiven Sitzungen angezeigt werden, über die er auch Sitzungen auf anderen Geräten abmelden kann. Dazu muss für Nutzer eine Zuordnung zwischen Sitzungen und Geräten herstellbar sein. Deshalb werden zu jeder Sitzung Informationen über das Gerät, an dem der Login erfolgte, mit abgespeichert und in der Auflistung angezeigt. Für Webapplikationen ist hierzu der User-Agent-String auswertbar, welcher vom Client als *HTTP*-Header zum Server übermittelt wird und Angaben zum Gerätemodell, Betriebssystem sowie *UA* enthält.

2.3.4 Zurücksetzung von Anmeldedaten

Sollte der Nutzer seine Zugangsdaten zum verifizierten Account vergessen, so muss er dennoch die Möglichkeit besitzen, ohne gültige Session Zugang zum Account zu erlangen. Zur Zurücksetzung des Passwortes wird eine Verifikation über den Hinterkanal durchgeführt, siehe Kapitel 2.3.1. Der Nutzer gibt im Client *mail* an und erhält eine Mail mit Zurücksetzungs-Token, der zur Neuvergabe eines Passworts ohne Angabe des alten Passworts berechtigt. Der Zurücksetzungs-Token kann nach Kapitel 2.2.4 clientgestützt mit folgenden Claims modelliert werden: Nutzerkennung, Gültigkeitszeitraum sowie Token-Typ „pw-reset“. Der Gültigkeitszeitraum sollte nicht zu kurz gewählt werden, da die Mail-Zustellung mit zeitlichem Verzug behaftet sein kann. Ein Nutzer erhält sonst möglicherweise nur abgelaufene Tokens und kann keine Zurücksetzung vornehmen.
Die Token-Signierung nutzt als Secret den Hashwert des bisherigen Passwortes. Dadurch wird ein Token invalidiert, sobald das Passwort geändert wurde, weshalb eine Token-Revocation entfällt. Allerdings können damit auch mehrere gültige Zurücksetzungs-Token zeitgleich existieren. Wenn einer dieser Tokens einem Dritten bekannt wird, so könnte dieser den Account übernehmen. Deshalb wird eine Nonce eingeführt, indem in der *DB* zu jedem Account der Zeitstempel der letzten Anforderung einer Zurücksetzung vermerkt wird. Dieser Zeitstempel wird

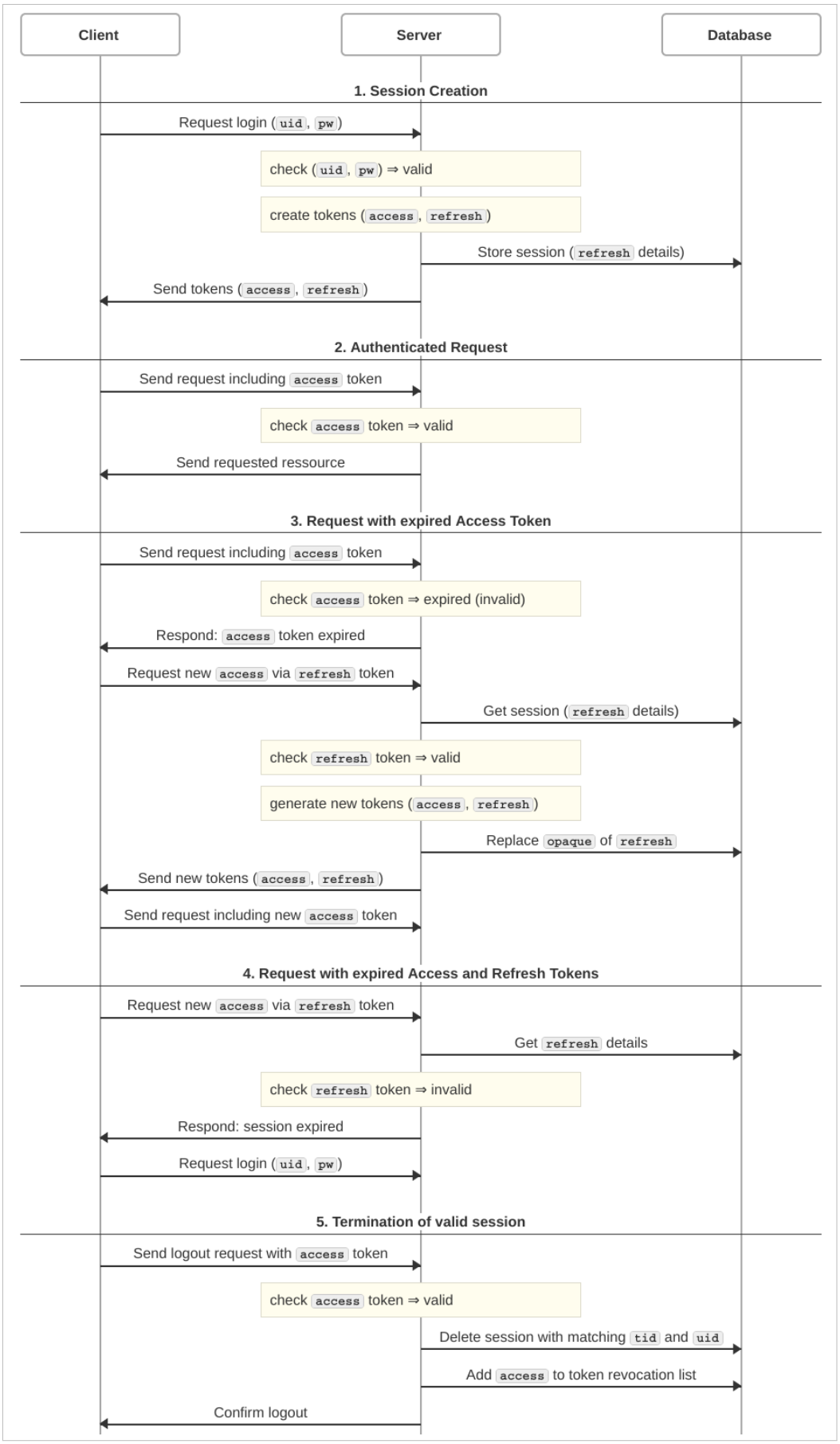


Abbildung 2.8: Abläufe der Sitzungsverwaltung im Referenz-Protokoll
Quelle: Eigene Darstellung

auch im Token vermerkt und beim Vorlegen eines Zurücksetzungs-Tokens durch den Server auf Gleichheit überprüft. Über den Zeitstempel kann zusätzlich ein Rate-Limiting implementiert werden, siehe Kapitel 4.2.3 und 5.1.5.
Nach erfolgreichem Zurücksetzen der Anmeldedaten sind die bestehenden Sitzungen des Nutzers zu invalidieren. Andernfalls könnte ein Angreifer, der z. B. dem Opfer sein mobiles Endgerät entwendet hat, eine auf dem Gerät etablierte Sitzung weiterhin nutzen und Schaden anrichten.

Auch diese Protokollfunktion ist anfällig für Messungen der Ausführungsdauer zur Bestimmung hinterlegter Mail-Adressen. Deshalb sollte der Mailversand zeitlich asynchron zur Serverantwort erfolgen. Die Dauer der *DB*-Abfrage zur Mail-Adresse darf nicht signifikant davon abhängen, ob die Adresse hinterlegt ist.

Die Zurücksetzung von Anmeldedaten ist besonders sicherheitskritisch, da der Zugriff auf die Applikation mit dem auf das Mail-Konto gleichgesetzt wird. Als Gegenmaßnahme werden mitunter Sicherheitsfragen als Ersatzfaktor zum Passwort herangezogen. Sie bieten jedoch kein adäquates Sicherheitsniveau. Die Antworten sind durch *OSINT* mitunter trivial zu ermitteln.[MM18, Rab08] Stattdessen sollte eine *MFA* etabliert werden, siehe Kapitel 4.

2.3.5 Änderung von Anmeldedaten

Eine Änderung des Passwortes muss auch möglich sein, ohne eine Zurücksetzung anzufordern. Dies geschieht im Kontext einer Sitzung. Der Nutzer gibt sein bisheriges pw_{old} an und wählt ein neues pw_{new}. Die Werte werden übermittelt und bei Korrektheit von Session sowie pw_{old} wird der Passwort-Hashwert in der *DB* aktualisiert. Der Session-Token allein verfügt nicht über die notwendige Berechtigungsstufe zur Passwort-Änderung. Diese wird durch Angabe von pw_{old} erlangt. Sonst wäre ein Angreifer mit gestohlenem Session-Token in der Lage, den Account-Besitzer auszusperren, ebenso wie durch Kenntnis von *uid* und *pw*. Der Besitzer könnte sich in diesem Szenario durch Zurücksetzung der Anmeldedaten wieder Zugang verschaffen, siehe Kapitel 2.3.4. Damit für das Opfer einer Accountübernahme das Geschehen transparent wird, sollte eine Passwort-Änderung via Mail-Nachricht dokumentiert werden. Eine Passwort-Änderung via Mail-Verifikation zu autorisieren, bietet keinen Sicherheitsvorteil, da sie durch Passwort-Zurücksetzung übersteuerbar ist.

Die Authentifizierung mittels Session-Token und Passwort lässt sich auch zur Autorisierung weiterer Prozesse mit hoher Kritikalität innerhalb der Applikation nutzen, z. B. zur Auslösung einer Bank-Überweisung.

Ebenfalls ist zum Schutz vor *Data Breaches* eine symmetrische Verschlüsselung sensitiver Nutzerdaten in der *DB* mittels Passwort möglich. Diese werden vom Server nur unter Nachweis von Passwort und Session ausgehändigt. Für den Fall des Passwortverlustes gehört ein zusätzlicher Backup-Schlüssel eingeführt.

Eine Passwort-Änderung kann durch den Nutzer angefordert oder durch eine Passwort-Richtlinie vorgegeben werden, z. B. als Policy zum Passwortwechsel nach 90 Tagen. Derartige Policies sollen den Zeitraum zur Nutzung entwendeter Credentials verringern. Jedoch sind sie kritisch zu sehen, da sie Nutzer zur Wahl einer abgewandelten Variante des alten Passwortes verleiten, z. B. durch Anhängen oder Hochzählen einer Zahl. Ein Angreifer kann mögliche Varianten leicht aus dem alten Passwort ableiten. Als Gegenmaßnahme kann eine Regel zur maximal erlaubten Ähnlichkeit von pw_{old} und pw_{new} implementiert werden, was jedoch der *UX* schadet und Nutzer zum Ankämpfen gegen diese Regel provoziert. Zum Verzicht auf die Policy kann eine verpflichtende *MFA* eingeführt werden.

2.3.6 Löschung von Accounts

Zur Löschung eines Accounts als Aktion mit hohem, irreversiblem Schadenspotenzial muss eine Mail-Verifikation erfolgen, siehe Kapitel 2.3.1. Zusätzlich kann eine Zeitverzögerung von z. B. 48 Stunden bis zur Ausführung der Aktion eingesetzt werden, um einen Abbruch zu ermöglichen. Konzeptionell kann die Token-Verwendung äquivalent zur Zurücksetzung aus Kapitel 2.3.4 implementiert werden. Um die Löschung vornehmen zu können, sind eine gültige Mail-Verifikation sowie Session nachzuweisen und der Lösch-Antrag im *UI* zu bestätigen. Mit Account-Löschung sind auch die gültigen Nutzersitzungen zu invalidieren. Abschließend sollte der Nutzer zur Dokumentation über die durchgeführte Account-Löschung per E-Mail benachrichtigt werden.

Von einer vollständigen Löschung sollte abgesehen werden, sofern die *uid* außenwirksam verwendet wurde. Ein Angreifer könnte sonst einen gelöschten Account erneut registrieren und unter der Identität des alten Account-Inhabers agieren. Aufgrund dieser möglichen Verwechslungsgefahr sollte bei Nutzerkennungen auch nicht zwischen Groß- und Kleinschreibung unterschieden werden.[5]
Um die betreffende *uid* registriert zu halten, kann ein zu löschender Account dauerhaft deaktiviert werden. Dann gilt es, die mit dem Account verknüpften Angaben entsprechend der Datenschutzvorgaben zu löschen oder zu verfremden.

[5] Das sich hieraus ergebende Problemfeld des homografischen Phishings und Typosquattings wird in Kapitel 4.2.2 behandelt.

3 Methoden clientseitiger Passwortverarbeitung

Clientseitige Passwortverarbeitung ist ein wiederkehrendes Diskussionsthema in Entwickler-Communities. Mitunter wird vom Entwurf eines solchen Protokolls abgeraten, da sich die Anwendungskomplexität erhöht. Zudem werden Zuständigkeiten zwischen Client und Server verschoben, weshalb zusätzliche Sicherheitsaspekte zu beachten sind. Daher werden nachfolgend verschiedene Protokollvarianten mit clientseitiger Passwortverarbeitung entwickelt und die jeweiligen Schwachstellen benannt sowie einem Threat-Modelling unterzogen. Als Techniken kommen clientseitiges Hashing in Kombination mit Challenge-Response-Verfahren zum Einsatz. Betrachtungsgegenstand sind die gegenüber dem Referenz-Protokoll anzuwendenden Änderungen und ihre Auswirkungen.

3.1 Architektonische Probleme

Die Umverteilung der Aufgaben zur Passwortverarbeitung zwischen Client und Server birgt architektonische Herausforderungen, wie nachfolgend thematisiert.

3.1.1 Rollenverteilung von Client und Server

Ausschließliches Client-Hashing

In einem ersten naiven Ansatz wird das Passwort-Hashing vom Server zum Client verlagert. Das Backend wird entlastet, da es keine Hashing-Operationen mehr durchführen muss. Zur Registrierung gibt der Nutzer uid und pw an. Der UA berechnet $pw' := hash_{cli}(pw)$ und überträgt (uid, pw') an den Server. Zur Anmeldung berechnet der UA gleichermaßen pw' und übermittelt (uid, pw'). Der Server mus lediglich die Gleichheit mit dem gespeicherten Hashwert überprüfen.
Dem Verfahren fehlen die in Kapitel 2.2.3 vorgestellten Sicherheitsmerkmale von *salt*, *pepper* und *cost*. Ihre clientseitige Integration ist Teil der weiteren Ausführungen. Wären die Sicherheitsmerkmale vorhanden, so würde das Verfahren einen wirksamen Schutz gegen reguläre Online-Zugriffe bieten. Wäre das Threat-Model auf diese beschränkt, würde im Kombination mit Rate Limiting das Schutzniveau des Referenz-Protokolls erreicht.
Das Schutzniveau wird sogar leicht verbessert, da der Server nie Klartext-Passwörter erfährt. Eine versehentliche Ausgabe von Klartext-Passwörtern am Server durch Implementierungsfehler, z. B. durch zum Debugging eingefügte und dann vergessene Log-Nachrichten, ist damit von vornherein ausgeschlossen.

Client-Hashing mit Server-Hashing

Allerdings schützt der skizzierte Ansatz in keiner Weise vor irregulären Zugriffen, wie in Kapitel 2.2.3 erläutert. Der Wert pw' übernimmt effektiv die Rolle des Klartextpasswortes. Damit ist das Verfahren anfällig für *Pass-the-Hash-Angriffe*, da zur Authentifizierung nur pw' und nicht pw bekannt sein muss. Ein Angreifer kann pw' aus der *DB* auslesen, da der Wert dort im Klartext persistiert ist. Trotz clientseitigem bleibt ein serverseitiges Hashing damit weiterhin obligatorisch.
Jedoch besitzt die Verteilung der Hashing-Last auf Client und Server das Potenzial, die Kostenanforderung an das Server-Hashing zu senken und damit den Server zu entlasten. Deshalb wird das Verfahren auch *Server-Relief* genannt. Ein Hashing mit *Server-Relief* kann zu folgenden Betriebsmodi ausgestaltet werden:

- **Statisch**: Client und Server leisten jeweils einen konstant vorgegebenen Hashing-Aufwand. Dabei können von Client und Server unterschiedliche Hashfunktionen eingesetzt werden, da sie nicht aneinander gekoppelt sind.
- **Dynamisch**: Der Hashing-Aufwand wird in Abhängigkeit von Server-Auslastung und Rechenkapazitäten des Client-Endgeräts aufgeteilt. Beispielsweise könnte der Client solange rechnen, bis er fertig oder ein Timeout erreicht ist. Ist mit dem Timeout noch nicht die Mindestvorgabe an Hash-Operationen erreicht, so muss der Client bis zu ihrer Erreichung weiterrechnen. Nach Übermittlung übernimmt der Server das übrige Hashing.

Der dynamische Modus bietet eine potenziell bessere Unterstützung für leistungsschwache Endgeräte, wenn der Server den Hauptteil der Hashinglast übernimmt. Im Falle eines hohen Lastaufkommens, z. B. aufgrund einer *DDOS*-Attacke, kann der Server schlicht die Mindestvorgabe an durch Clients zu leistende Hash-Operationen hochsetzen. Jedoch muss der Server, wie oben erläutert, dennoch weiterhin ein minimales Hashing leisten.
Allerdings ist der dynamische *Server-Relief*-Modus nur mit speziellen Hash-Funktionen durchführbar, die nativ eine Aufteilung von Hashing-Operationen unterstützen. Hierzu zählen z. B. Catena und Rig.[CJMS14] Klassische, etablierte und erprobte Hashfunktionen, z. B. Bcrypt oder Argon2, sehen in der Regel keinen derartigen Betriebsmodus vor.[FLW13, S. 6] Dieser könnte nachträglich hinzugefügt werden, wozu aber interne Zustände und Algorithmen der Hashing-Verfahren zu exponieren wären. Dies sicher zu gestalten, liegt nicht im Rahmen dieser Thesis. Stattdessen werden die Hashing-Verfahren als geschlossene Einheiten betrachtet. Um eine hohe Interoperabilität und bestmögliche Kompatibilität mit bestehenden, anerkannten Verfahren zu ermöglichen, wird deshalb nachfolgend das statische *Server-Relief* betrachtet.

Jedoch wären prinzipiell auch sämtliche in diesem Kapitel vorgestellten Konzepte ohne Nachteil in der dynamischen Variante umsetzbar. Die Sicherheitsbetrachtung muss dann für die Ränder möglicher Partitionierungen erfolgen, also für maximal aufwändiges Client- mit minimalem Server-Hashing und umgekehrt.

Die Ausgestaltung der statischen Client-Server-Aufteilung zum Hashing ist Gegenstand von Kapitel 3.2. Das Problem der *Pass-the-Hash-Angriffe* wird erst in Kapitel 3.3 durch Einführung von Challenge-Response-Verfahren gelöst.

3.1.2 Verhinderung unsicherer Passwörter

Passwortregeln

Ein weiteres Problemfeld stellt die Verhinderung einer Wahl schwacher Passwörter bei Registrierung, Änderung oder Zurücksetzung von Anmeldedaten dar. Hierzu werden Regeln zur Komplexität eingesetzt, z. B. zur Mindestlänge, Einsatz von Ziffern sowie Sonderzeichen oder auch Verbote bestimmter Muster. Diese Passwortregeln sollten durch den *IDP*-Betreiber vorgegeben werden, um eine untere Schranke für die Stärke eingesetzter Passwörter festzulegen.[Poh19]
Die definierten Regeln müssen bei Passwortvergabe gegenüber dem Nutzer durchgesetzt werden. Nur der Server kann eine Regeldurchsetzung garantieren, da er bei Regelverstoß die Bearbeitung einer Anfrage ablehnen kann. Der Client hat lediglich die Möglichkeit, den Nutzer auf einen Regelverstoß hinzuweisen. Jedoch kann sich der Nutzer darüber hinwegsetzen, da er den *UA* kontrolliert.

Bei Durchführung eines Client-Hashings wird dem Server die Fähigkeit der Regeldurchsetzung genommen, da er keine Kenntnis über den Passwort-Klartext erlangt. Deshalb muss der *UA* die Regelumsetzung gegenüber dem Nutzer einfordern. Da der Nutzer sich mit gewissem Aufwand darüber hinwegsetzen kann, muss der *IDP*-Betreiber eine Abwägung treffen: Einerseits sollen die Regeln eine möglichst hohe Passwort-Komplexität garantieren. Andererseits muss der Aufwand für die Einhaltung der Regeln niedrigschwellig genug angesetzt sein, sodass das Widersetzen dagegen für einen Nutzer mit deutlich höherem Aufwand verbunden wäre. Ein Nutzer, der vorsätzlich gegen die Sicherheitsvorgaben des Anwendungsbetreibers verstößt, kann im Zweifelsfall nicht an der Wahl eines unsicheren Passwortes gehindert werden.
Als Szenario wäre bei Webanwendungen z. B. die Entwicklung einer Browser-Erweiterung zur Umgehung der Passwortregeln durch einen technisch versierten Anwender denkbar. Wenn diese Browser-Erweiterung in Umlauf käme, hätten auch weniger versierte Nutzer eine aufwandsarme Umgehungsmöglichkeit.

In einer grafischen *UI* können die Passwortregeln interaktiv gegenüber dem Nutzer eingefordert werden. Wird stattdessen lediglich die durch die Applikation exponierte *API* über ein Drittprogramm konsumiert, so muss dieser *UA* ebenfalls die Passwortregeln implementieren. Für eine solche Externalisierung sollten den Entwicklern der Drittprogramme Dokumentation oder auch vorgefertigte Programm-Bibliotheken bereitgestellt werden. Eine Regeldurchsetzung ist auch hier nicht möglich. Ebenso kann der *IDP*-Betreiber nicht kontrollieren, ob Drittprogramme die Regeln implementieren. Soll eine Regeländerung stattfinden, so muss diese auch an die Drittprogramme propagiert werden.

Eine weitere Regel betrifft den Prozess der Passwort-Änderung. Wie in Kapitel 2.3.5 erläutert, kann gefordert werden, dass pw_{old} und pw_{new} hinreichend unterschiedlich sein müssen. Wird das Passwort clientseitig mit Salt und Pepper gehasht, kann das Backend nicht mal mehr die Gleichheit von pw_{old} und pw_{new} feststellen. Deshalb ist die Überprüfung zur Passwort-Ähnlichkeit ebenfalls durch den *UA* durchzuführen und unterliegt den gleichen Problemen.

Bekannte Passwörter

Zusätzlich zu den Passwortregeln sollte überprüft werden, dass das gewählte Nutzerpasswort nicht auf einer Liste bekannter Passwörter vorhanden ist. Diese Listen entstehen u. a. durch *Data Breaches*. Die enthaltenen Passwörter sind als sehr schwaches Schlüsselmaterial zu bewerten, da sie bevorzugt in Wörterbücher für Angriffe aufgenommen werden.
Ein serverseitiges Vorhalten einer entsprechenden Liste erlaubt die Überprüfung eines Passwortes zum Vorkommen auf der Liste. Dies impliziert jedoch eine Passwort-Übertragung an das Backend im Klartext, welche es zu verhindern gilt.
Stattdessen könnte die Liste komprimiert an den Client übertragen werden, um dort die Überprüfung durchzuführen. Allerdings skaliert dieser Ansatz schlecht, da nur begrenzte Datenmengen performant übertragen und durch den Client dekomprimiert sowie geparst werden können. Sei z. B. eine Liste pw^* mit $|pw^*| = 5 * 10^8$ Passwörtern der Länge $|pw| = 8$ gegeben. Bei einem Encoding von einem Byte je Zeichen zuzüglich Trennzeichen ergibt sich eine unkomprimierte Datenmenge von $(8 + 1) * 5 * 10^8 \approx 4,2GiB$. Mobile Geräte könnte die Übertragung und leistungsschwache Geräte die Verarbeitung dieser Datenmenge überfordern. Die Verwendung einer kürzeren Liste würde die Stärke der zugelassenen Passwörter senken. Aufgrund mangelnder Skalierbarkeit wird von dem Ansatz abgesehen.

Stattdessen kann ein Backend-Dienst involviert werden, ohne das Passwort an diesen zu exponieren: Der Client bildet mit einer schnellen Hashfunktion $hash^{fast}_{check}$

den Hashwert von pw. An den Server wird nur das Präfix des Hashwertes in Form der ersten a Hashbits übertragen: $prefix := hash_{check}^{fast}(pw)[0..a]$. Der Server verfügt über eine mittels $hash_{check}^{fast}$ gehashte, sortierte Liste schwacher Passwörter. Von dieser gibt er die Menge mit übereinstimmendem Präfix zurück. Der Client kann nun prüfen, ob $hash_{check}^{fast}(pw)$ in der Server-Antwort enthalten ist.
Das angewendete Prinzip wird als k-Anonymität bezeichnet. Es werden $k := |hash_{check}^{fast}| - a$ Bits des Hashwertes gegenüber dem Backend anonym und damit geheim gehalten. Durch das Verfahren erhält der Server hinreichend genaue Angaben über das Passwort, um eine sinnvoll begrenzte Antwortmenge zu erzeugen. Da er nicht erfährt, ob das gesuchte Passwort auf der Antwortliste enthalten war, kann er es nicht direkt ermitteln.[Hun18]
Allerdings könnte ein Angreifer mit Server-Zugriff versuchen, das Passwort pw_v des Opfers v zu bestimmen. Dazu beobachtet er den Registrierungsprozess von v am Server und sammelt folgende Informationen:

- pw_v'': Der durch Anwendung von $hash_{cli}^{slow}$ und $hash_{svr}^{slow}$ erzeugte Hashwert, welcher zum Nutzer in der DB gespeichert ist.
- $prefix_v$: Das mithilfe von $hash_{check}^{fast}$ erzeugte Präfix $prefix_v$ zu pw_v.

Zur Bestimmung von pw_v versucht der Angreifer einen Dictionary-Angriff. Für jeden Passwort-Kandidaten kann er durch Anwendung von $hash_{check}^{fast}$ eine Übereinstimmung mit $prefix_v$ prüfen. Nur für passende Kandidaten wird das ressourcenintensive Hashing durch $hash_{cli}^{slow}$ und $hash_{svr}^{slow}$ ausgeführt. Durch diese Vorfilterung kann der Angreifer den Aufwand der Attacke verringern.
Um das Passwort geheim zu halten, ist die Präfix-Länge deshalb so kurz wie möglich zu wählen. Allerdings muss der Wert groß genug gewählt werden, sodass die Ergebnismenge einer Abfrage eine verarbeitbare Größe behält. In Kapitel 5.1.4 wird als Teil der Implementierung eine konkrete Präfix-Länge bestimmt.
In einem zweiten Szenario könnte der Angreifer nicht nur einen Präfix-Wert, sondern eine zeitlich zusammenhängende Sequenz von Präfix-Anfragen eines bestimmten Nutzers aufgezeichnet haben. Dann kann der Angreifer annehmen, dass die vorhergehenden Anfragen der Sequenz schwachen Passwörtern mit ähnlichen Bestandteilen zuzuordnen sind. Zur Rekonstruktion kann der Angreifer zu den Präfixen passende, schwache Passwörter finden, welche Variationen zueinander darstellen. Nun kann der Angreifer ein nutzerspezifisches Wörterbuch mit nichtschwachen Passwörtern aufbauen, indem er die gefundenen Variationen mutieren lässt. Die Mutation einer Passwort-Zeichenkette kann durch Anwendung bekannter Muster zum Einfügen, Ändern oder Vertauschen von Zeichen erfolgen, z. B. durch Anhängen einer Jahreszahl.
Um dieses Szenario zu erschweren, sollten die Präfix-Werte als Geheimnisse ange-

sehen und z. B. nicht in Log-Daten von Backend-Komponenten gespeichert werden. In Kapitel 5.1.4 folgt die Betrachtung zur praktischen Umsetzung.

Abbildung 3.1 fasst das erläuterte Vorgehen zur Umsetzung der Passwortregeln und Verhinderung schwacher Passwörter anhand des Prozesses zur Änderung von Anmeldedaten zusammen.

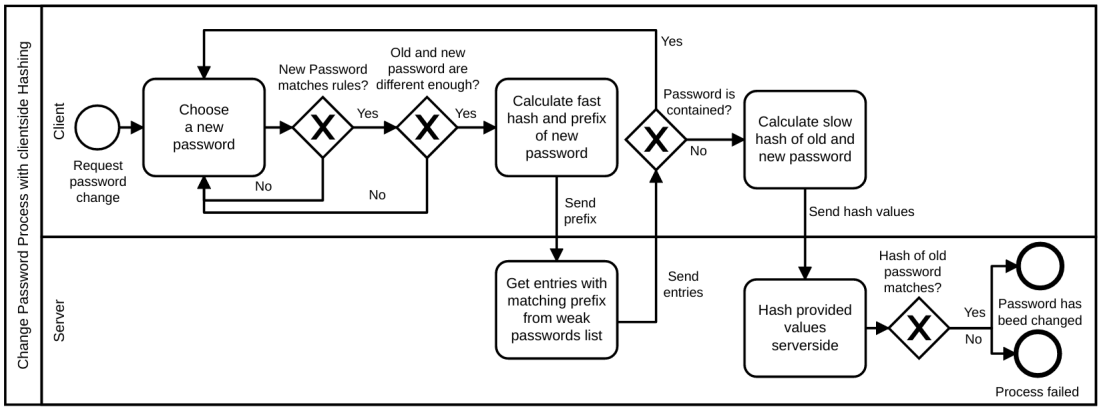


Abbildung 3.1: Vorgehen zur Wahl starker Passwörter ohne Passwortkenntnis durch den Server anhand des Prozesses zur Änderung von Anmeldedaten
Quelle: Eigene Darstellung

3.1.3 Risiko-Bewertung

Nachfolgend wird eine Risiko-Bewertung zu den architektonischen Herausforderungen clientseitiger Passwortverarbeitung vorgenommen. Als Methodik zum Threat-Modelling wird das in Kapitel 2.1.3 erläuterte DREAD-D eingesetzt.

Das Passwort-Hashing dient, wie in Kapitel 2.2.3 erläutert, einzig dem Schutz vor irregulären Zugriffen, wie *Data Breaches*. Zur Bewertung der möglichen Bedrohungen muss danach unterschieden werden, ob der Angreifer auch die Pepper kennt. Wie in Tabelle 3.1 und 3.2 aufgeführt, macht erst eine Pepper-Kenntnis durch den Angreifer das Szenario besonders gefährlich.

Im Referenz-Protokoll wurde das Passwort im Klartext an den Server übertragen. Wie in Tabelle 3.3 erkennbar, senkt der *TLS*-Einsatz hierbei das Risiko. Aufgrund der Pass-The-Hash-Problematik gilt die Betrachtung gleichermaßen für einfache Passwort-Hashes. Erst durch Einsatz von Challenge-Response-Verfahren, siehe Tabelle 3.4, kann das Risiko weiter gesenkt werden, vergleiche Kapitel 3.3.

Ein ausschließliches Client-Hashing bietet keinen Schutz gegen irreguläre Zugriffe, woraus das hohe Scoring in Tabelle 3.5 resultiert. Im Gegensatz dazu ist ein ergänzendes Client-Hashing zum Server-Hashing, wie erläutert, architektonisch unbedenklich, siehe Tabelle 3.6.

Tabelle 3.1: Risiko durch Angreifer mit DB-Einblick ohne Peppers

D	1/5	Kenntnis der Passwort-Hashwerte
R	1/5	Aufwändiges Hashing ohne Pepper-Kenntnis
E	2/5	Zugang zu DB-Auszug notwendig
A	5/5	Angriff betrifft sämtliche Accounts
$\sum$	**9/20**	$\Rightarrow$ **Niedriges Risiko**

Tabelle 3.2: Risiko durch Angreifer mit Einblick in DB und Peppers

D	5/5	Übernahme von Accounts
R	2/5	Aufwändiges Hashing mit Peppers
E	1/5	Zugang zur Server-Infrastruktur
A	5/5	Angriff betrifft sämtliche Accounts
$\sum$	**13/20**	$\Rightarrow$ **Mittleres Risiko**

Tabelle 3.3: Risiko der Übertragung von Passwörtern und Hashwerten via TLS an den Server

D	5/5	Impersonation für abgefangene Credentials möglich
R	5/5	Abfangen bei jeder Authentifizierung mit Passwort möglich
E	1/5	Einbruch in TLS-Verbindung notwendig
A	1/5	Nur Nutzer kompromittierter Verbindungen betroffen
$\sum$	**12/20**	$\Rightarrow$ **Mittleres Risiko**

Tabelle 3.4: Risiko einer Authentifizierung nach Challenge-Response via TLS

D	5/5	Bestimmung des übertragenen Passwortwertes
R	1/5	Abfangen übertragener Werte und aufwändiges Hashing nötig
E	1/5	Einbruch in TLS-Verbindung notwendig
A	1/5	Nur Nutzer kompromittierter Verbindungen betroffen
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Eine *DDOS*-Attacke kann die Verfügbarkeit des Backends einschränken und damit legitimen Nutzern den Zugang zur Anwendung verwehren. In Tabelle 3.7 und 3.8 wird die Anfälligkeit zur Ressource Exhaustion von aufwändigem und aufwandsarmen Server-Hashing gegenübergestellt. Wie aus dem Vergleich hervorgeht, ist ein *Server-Relief* anzustreben.

Aus dem Client-Hashing folgt eine Umgehungsmöglichkeit für Passwortregeln und dem Verbot bestimmter Passwörter. Wie in Tabelle 3.9 aufgeführt, ist die Bedrohung niedrig: Sie betrifft nur Nutzer, welche die Regeln umgehen. Zudem wird der unsichere Client-Hashwert weiterhin durch das Server-Peppering geschützt. Im Gegensatz dazu betrifft die Präfix-Übertragung auch regelkonform agierende Nutzer. Wie in Tabelle 3.10 dargestellt, ist die stark begrenzte Informationspreisgabe zum Schlüsselmaterial tolerierbar, da Transport und serverseitige Verarbeitung geschützt erfolgen.

Auf Basis dieser Einschätzung ist eine sichere clientseitige Passwortverarbeitung mittels Client-Hashing architektonisch umsetzbar.

Tabelle 3.5: Risiko von ausschließlichem Client-Hashing

D	5/5	Account-Übernahme
R	5/5	Auslesen der Hash-werte aus der DB
E	2/5	Zugriff auf DB notwendig
A	5/5	Betrifft sämtliche Accounts, Angriff parallelisierbar
$\sum$	**17/20**	$\Rightarrow$ **Hohes Risiko**

Tabelle 3.6: Risiko von Client-Hashing mit Server-Hashing

D	5/5	Account-Übernahme
R	1/5	Hoher Hashing-Aufwand
E	1/5	Zugriff auf DB und Server-Pepper notwendig
A	1/5	Betrifft jeweils einen Account
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Tabelle 3.7: Risiko der Überlastung für langsames Server-Hashing

D	3/5	Anwendung unbenutzbar solange der Angriff anhält
R	5/5	Angriff ohne technische Expertise möglich
E	3/5	Umgehung Rate-Limitings mit moderater Request-Anzahl
A	4/5	Ausfall des Backends betrifft potenziell viele Nutzer
$\sum$	**15/20**	$\Rightarrow$ **Hohes Risiko**

Tabelle 3.8: Risiko der Überlastung für schnelles Server-Hashing

D	2/5	Anwendung während Angriff eingeschränkt benutzbar
R	2/5	Angriff benötigt sehr hohes Anfragevolumen
E	2/5	Umgehung Rate-Limitings mit hoher Request-Anzahl
A	4/5	Ausfall des Backends betrifft potenziell viele Nutzer
$\sum$	**10/20**	$\Rightarrow$ **Mittleres Risiko**

Tabelle 3.9: Risiko zur Umgehung der Passwort-Regeln

D	4/5	Erleichtertes Erraten des Passwortes
R	1/5	Hoher Aufwand zum Hashing
E	1/5	Zugriff auf DB und Server-Pepper notwendig
A	2/5	Nur regeln-umgehende Nutzer betroffen
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Tabelle 3.10: Risiko zur Übertragung von Passwort-Hashpräfixen

D	3/5	Einschränkung des Passwort-Suchraums
R	2/5	Nur bei Wahl eines neuen Passwortes möglich
E	1/5	Einbruch in TLS-Verbindung oder Server-System
A	2/5	Präfix je Account, aber aus einer Grundgesamtheit
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

3.2 Clientseitiges Hashing von Passwörtern

Von den IDP-Funktionen betrachtet dieses Kapitel vorrangig die Registrierung und Anmeldung. Diese sind am besten geeignet, um die Veränderungen am Protokoll zu erläutern. Nachfolgend werden die Schritte zur Etablierung und Absicherung von Client-Hashing vorgestellt.

3.2.1 Angriffe auf Client-Hashing ohne Salt oder Pepper

Zur clientseitigen Hasherzeugung aus Passwörtern durch den *UA* soll eine langsame Hashfunktion $hash_{cli}^{slow}$ ergänzend zum serverseitigen Hashing durch $hash_{svr}^{slow}$ eingeführt werden.
Für die Registrierung berechnet der Client $pw' := hash_{cli}^{slow}(pw, cost_{cli})$ und übermittelt pw' sowie uid an den Server. Dieser führt das serverseitige Hashing durch: $pw'' := hash_{svr}^{slow}(salt_{svr} \parallel kdf_{svr}(pepper_{svr}, uid) \parallel pw', cost_{svr})$
Der Wert pw'' wird in der *DB* gespeichert. Zur Anmeldung erfolgt das Hashing äquivalent. Das Verfahren ist als simple Ergänzung zum Referenz-Protokoll realisierbar, wobei lediglich die Client-Implementierung anzupassen ist.
Zusätzlich kann die Server-Validierung für Request-Parameter verschärft werden: Im Gegensatz zu pw besitzt pw' eine durch $|hash_{cli}^{slow}|$ fest vorgegebene Länge, z. B. 256 $Bits$. In [OWA21] wird bezüglich der Passwortlänge $|pw|$ die Empfehlung $8 \leq |pw| \leq 64$ ausgesprochen. Ein Passwort aus *Unicode*-Zeichen mit bis zu 4 Bytes je Zeichen kann dann maximal $64 * 4 * 8 = 2048\,Bits$ umfassen. Client-Hashing verkleinert damit den Angriffsvektor für *DDOS*-Attacken, da die Eingabedatenlänge für das Server-Hashing geringer ausfällt.

Aus dem oben skizzierten Protokoll ergeben sich die nachfolgend erläuterten Angriffsvektoren gegen die Hashing-Operationen:

1. Angriff auf pw'' zur Bestimmung von pw
2. Angriff auf pw'' zur Bestimmung von pw'
3. Angriff auf pw' zur Bestimmung von pw

Angriff auf pw″ zur Bestimmung von pw

Dieser ist der zum Referenz-Protokoll äquivalente Angriff auf den in der *DB* hinterlegten Hashwert. Der Angreifer kennt einen Wert pw'' und möchte das zugehörige pw ermitteln. Zur Bestimmung von pw'' sind sowohl $hash_{svr}^{slow}$ als auch $hash_{cli}^{slow}$ zu berechnen. Bei konstantem $cost_{svr}$ ist der Berechnungsaufwand damit größer als beim Referenz-Protokoll.
Das Fehlen der Sicherheitsmerkmale $salt_{cli}$ und $pepper_{cli}$ zum Client-Hashing

birgt kein Sicherheitsproblem, solange $salt_{svr}$ und $pepper_{svr}$ intakt sind und $cost_{svr}$ hoch gewählt ist. Ein Angreifer kann eine generische Hashtable für $hash_{cli}^{slow}$ aufbauen. Ihre Einträge muss er aber dennoch separat $hash_{svr}^{slow}$ zuführen. Der Vorteil gegenüber dem Referenz-Protokoll liegt damit im Aufwand zum Aufbau der Hashtable.[ith15]

Die Gesamtkosten der Hashing-Operationen ergeben sich aus den Werten für $cost_{svr}$ und $cost_{cli}$. Allerdings würde eine Addition beider Kostenvektoren aus dem eben erläuterten Grund zu kurz greifen. Wenn $salt_{cli}$ und $pepper_{cli}$ eingeführt würden, könnte der Angreifer trotzdem versuchen das Hashing aufzuteilen: Seien beide Hashfunktionen rechenintensiv aber nur $hash_{svr}^{slow}$ speicherintensiv. In diesem Szenario könnte $hash_{cli}^{slow}$ auf spezialisierter Hardware mit hoher Durchsatzrate berechnet werden, siehe Kapitel 2.2.3. Die erzeugten Hashwerte könnten einer separaten Infrastruktur zur Berechnung von $hash_{svr}^{slow}$ zugeführt werden. Ein Angreifer könnte die Kosten optimieren, da für den Angriff zwei verschiedene Infrastrukturen in unterschiedlicher Skalierung verwendbar sind. Deshalb sollten die Gesamtkosten nicht aus zwei schwachen Kostenvektoren zu einem starken zusammengesetzt werden. Stattdessen gilt es durch zumindest einen der Teilnehmer (Client oder Server) einen Kostenvektor zu wählen, der hinreichend rechen- als auch speicherintensiv ist. Angenommen dies gelte für $cost_{cli}$ und $salt_{cli}$ sowie $pepper_{cli}$ würden verwendet. Dann wäre es für diesen Angriff unerheblich, wie stark das Server-Hashing ist. Da beide Hashfunktionen auszuführen sind, muss der Server nicht die Hauptlast vom Hashing tragen. Voraussetzung bleibt ein kooperativer Nutzer, der $hash_{cli}^{slow}$ wie angefordert ausführt.

Angriff auf pw″ zur Bestimmung von pw′

Dieser Ansatz zielt auf die Möglichkeit zum *Pass-the-Hash-Angriff* auf das Protokoll ab: Da am Server nur pw' und nicht pw vorzulegen ist, muss ein Angreifer pw nicht kennen, um sich authentifizieren zu können. Als Hashing-Operation ist für den Angriff wie beim Referenz-Protokoll nur $hash_{svr}^{slow}$ anzuwenden. Allerdings wird durch die clientseitige Ableitung von pw' aus pw das Schlüsselmaterial gestreckt. Damit ist die Eingabe für $hash_{svr}^{slow}$ kein niederentropisches Passwort pw sondern ein hochentropischer Hashwert pw'. Ein gezielter Dictionary-Angriff im Definitionsraum von $hash_{cli}^{slow}$ (mit Werten für pw) wird zu einer Brute-Force-Attacke auf den Werteraum von $hash_{cli}^{slow}$ degradiert (mit Werten für pw').

Zunächst wird pw' rein aus pw abgeleitet (ohne $salt_{cli}$ oder $pepper_{cli}$). Damit bleibt pw' anfällig für Angriffe durch vorberechnete Hashwerte, siehe Kapitel 2.2.3. Der Angreifer verwendet hierzu eine Hashtable für typische Passwörter. Die Hash-

werte der Tabelle fungieren als Kandidaten für den Wert pw'. Durch Ausführung von $hash_{svr}^{slow}$ auf die Kandidaten ist bei Fund das zugehörige Passwort der Hashtable entnehmbar. Ein Angreifer besitzt damit eine schnellere Angriffsmöglichkeit als die erschöpfende Suche im Werteraum von $hash_{cli}^{slow}$. Durch Integration der Sicherheitsmerkmale $salt_{cli}$ und $pepper_{cli}$ in das Client-Hashing wird dieses Problem mitigiert. Erst dann ist ein Angreifer, wie gewünscht, gezwungen pw' aufgrund seiner Bitlänge als hochentropischen Wert zu betrachten. Die Bestimmung eines pw' aus pw'' erfordert damit eine erschöpfende Suche im Werteraum von $hash_{cli}^{slow}$. Diese Angriffsform wird nun betrachtet.

Um eine erschöpfende Suche im Werteraum von $hash_{cli}^{slow}$ möglichst aufwändig zu gestalten, muss $|hash_{cli}^{slow}|$ hinreichend hoch gewählt werden. Über diesen Parameter wird sichergestellt, dass ausgehend von pw'' der Angriff auf pw' ineffektiver ist als der auf pw. Zur Demonstration soll beispielhaft die Größe des Suchraums eines starken Passwortes dem eines Hashwertes gegenübergestellt werden:
Sei $hash_{cli}^{slow} = bcrypt$ gegeben, wobei $|bcrypt| = 184$ Bits gilt. Sei weiterhin Σ ein Alphabet mit $|\Sigma| = 100$ und $pw \in \Sigma^*$ mit $8 \leq |pw| \leq 20$. Der Suchraum für pw hat die Dimension $\sum_{i=8}^{20} 100^i \approx 1,2 * 10^{39}$. Der Suchraum für pw' umfasst $2^{184} \approx 2,5 * 10^{55}$ Werte. Er liegt damit um Größenordnungen über dem von pw.

Neben der Suchraumgröße muss auch der Zeitaufwand betrachtet werden. Seien t_{cli} und t_{svr} als durchschnittliche Ausführungsdauer von $hash_{cli}^{slow}$ sowie $hash_{svr}^{slow}$ gegeben. Sei pw^* der gewählte Suchraum für pw. Sei pw'^* der Suchraum für pw', wobei $|pw'^*| = 2^{|hash_{cli}^{slow}|}$ gilt. Nun soll der Zeitaufwand für eine erschöpfende Suche nach folgenden Angriffsvektoren verglichen werden:

- Angriff auf pw'' zur Bestimmung von pw (vorheriger Abschnitt):
  $t_1 = |pw^*| * (t_{cli} + t_{svr})$
- Angriff auf pw'' zur Bestimmung von pw' (dieser Abschnitt):
  $t_2 = |pw'^*| * t_{svr}$

Die Hashing-Verkettung aus Client- und Server-Hashing darf die Sicherheit nicht senken, weshalb pw das schwächste Glied sein muss. Deshalb soll der Angriff gegen das verarbeitete Passwort pw' schwerer sein als gegen das originale pw. Damit soll $t_1 < t_2$ gelten. Ausgehend vom vorherigen Beispiel sei $|pw^*| = 10^{39}$ und $|pw'^*| = 10^{55}$. Wenn $t_{cli} = t_{svr}$ sei, trifft $t_1 < t_2$ auf die Beispielwerte zu.
Nun soll der Wert $cost_{svr}$ und damit t_{svr} gesenkt werden, um den Server zu entlasten: Sei $t_{cli} = 1s$ und $t_{svr} = 1ns = 10^{-9}s$. Daraus ergibt sich $t_2 \approx 10^7 t_1$. Für einen Angreifer ist der Angriff in t_1 zeiteffektiver. Anzumerken ist, dass mit den Beispielwerten beide Angriffe nicht in praktikabler Zeit durchführbar sind.

Bezogen auf den betrachteten Angriff kann $cost_{svr}$ so niedrig gewählt werden,

dass $hash_{svr}^{slow} = hash_{svr}^{fast}$ gilt. Die kürzere Hashing-Zeit ermöglicht einen höheren Request-Durchsatz sowie eine bessere Skalierbarkeit.
Unter Einsatz von $salt_{cli}$ und $pepper_{cli}$ bleibt bei hinreichender Hashwertlänge, z. B. $|hash_{cli}^{slow}| = 256\,Bits$, der Angriff auf pw der erfolgversprechendste.

Für die *HMAC*-Signaturen der clientgestützten Bearer-Tokens aus Kapitel 2.2.4 genügt aus den erläuterten Gründen ebenfalls eine schnelle Hashfunktion. Zur Demonstration: Wenn die Secret-Länge 128 Bit und die durchschnittliche *HMAC*-Berechnungsdauer $10^{-12}s$ betragen, benötigt ein Angreifer mit 10^{10} parallelen Prozessoren $2^{128} * 10^{-12}/10^{10} \approx 10^9$ Jahre zur Signaturfälschung.

Angriff auf pw′ zur Bestimmung von pw

Dieser Angriff bedingt, dass der Angreifer pw' in Erfahrung bringen kann. Dazu muss er auf Seiten von Client, Server oder der Übertragung pw' unverschlüsselt mitlesen können. Ausgehend von der Threat-Analyse aus Kapitel 2.1.2 gilt es den Angriff auf die Übertragung zu betrachten. Ein *MITM*, der die *TLS*-Sicherheitsarchitektur umgangen hat, würde im Referenz-Protokoll bereits das Klartext-Passwort einsehen können.
Aufgrund der Anfälligkeit des Protokolls für *Pass-the-Hash-Angriffe* kann sich der Angreifer nach Kenntnisnahme von pw' direkt unter der Identität seines Opfers beim *IDP* anmelden. Das Client-Hashing schützt den Nutzer trotzdem in einem Aspekt: Falls der Nutzer das gleiche oder ein ähnliches Passwort für andere Applikationen wiederverwendet hat, kann der Angreifer nicht direkt zur Credential-Stuffing-Attacke übergehen. Stattdessen muss er zunächst pw' auf pw zurückführen. Damit dies eine Hürde darstellt, gilt es neben einem hohen Wert für $cost_{cli}$ die Sicherheitsmerkmale $salt_{cli}$ und $pepper_{cli}$ zu ergänzen. Diese Betrachtung folgt.

3.2.2 Client-Hashing mit Salt ohne Pepper

Zum Schutz gegen Angriffe mit vorberechneten Hashwerten soll ein Salt in das clientseitige Hashing eingestreut werden. Dabei erfolgt die Salt-Bildung äquivalent zur Pepper-Konstruktion des Referenz-Protokolls, siehe Kapitel 2.3.1. Die *uid* fließt in die Salt-Bildung ein, um das Salt an den Account zu binden:

$$salt'_{cli}(uid) := kdf_{cli}(salt_{cli}, uid)$$
$$pw' := hash_{cli}^{slow}(salt'_{cli}(uid) \parallel pw, cost_{cli})$$

Das $salt_{cli}$ wird als statischer Wert im Client-Programm hinterlegt. Entgegen gängiger Empfehlungen, wie [Hor19], sollte $salt_{cli}$ nicht aus Bestandteilen der

aufgerufenen *URL* der Webapplikation konstruiert werden. Dadurch würde eine vermeidbare Kopplung von Applikation und *Deployment* geschaffen. Wenn z. B. bei Änderung des Firmennamens die Webapplikation auf eine andere Domain umgezogen wird, könnten sich die Nutzer nicht mehr anmelden, wenn der Domainname zur Bildung von $salt_{cli}$ verwendet wurde. Stattdessen sollte $salt_{cli}$ einmalig durch einen *CSPRNG* erzeugt werden. Eine Implementierung dieses Authentifizierungsschemas könnte in mehreren Applikationen verwendet oder als Programm-Bibliothek angeboten werden. Dann sollte jede Anwendungsinstanz ein individuelles $salt_{cli}$ verwenden, um applikationsübergreifende Angriffe zu erschweren.

Das Referenz-Protokoll sah eine Anmeldung mit *uid* oder *mail* vor, siehe Kapitel 2.3.2. Da das Client-Salt aus *uid* abgeleitet wird, ist die Anmeldung nur noch mit der Nutzerkennung möglich. Wenn aus *UX*-Gründen auch eine Anmeldung via *mail* zugelassen werden soll, muss eine Fallunterscheidung vorgenommen werden, da am Client nur entweder *uid* oder *mail* vorliegt. Daraus resultieren für einen Account zwei Client-Salts, Client-Hashes, Server-Salts und Server-Hashes. In der *DB* müssen jeweils beide Werte je Account gespeichert werden. Die Dopplung erhöht die Komplexität der Implementierung und damit die Angriffsoberfläche. Sie ist jedoch aufgrund der im Referenz-Protokoll getroffenen Maßnahmen umsetzbar, ohne Auswirkungen auf die Protokoll-Sicherheit zu haben. Nachfolgend wird nur die Anmeldung via *uid* betrachtet.

Angriffe auf das Client-Salting

Die Angriffe gegen das Client-Salting leiten sich aus den in Kapitel 3.2.1 erläuterten Angriffen ab. Problematisch ist, dass ein Angreifer jederzeit aus dem Client-Quellcode den Wert $salt_{cli}$ extrahieren kann. Damit hat er potenziell eine große Vorlaufzeit zur Vorbereitung. Da $salt'_{cli}(uid)$ abhängig von Applikationsinstanz und Nutzerkennung ist, sind durch den Angreifer für jeden Account separate Vorbereitungen zu treffen. Ohne den Einsatz des Salts hätte der Aufbau einer generischen Rainbow-Table für den Algorithmus $hash_{cli}^{slow}$ genügt.

Ein Angreifer könnte annehmen, dass er zu einem späteren Zeitpunkt pw' des Opfers abfangen kann, siehe Kapitel 3.2.1. Deshalb könnte er eine Rainbow-Table gegen $hash_{cli}^{slow}$ mit $salt'_{cli}(uid)$ des Opfers aufbauen. Da $salt_{cli}$ nicht von einem Passwortwechsel beeinflusst wird, kann er die Rainbow-Table auch über einen längeren Zeitraum aufbauen und verwenden. Dagegen wäre eine Konstruktion zur Bindung des Passwortes an das Salt denkbar: $salt'_{wrong}(uid, pw) := kdf_{cli}(salt_{cli} \parallel pw, uid)$ Dieser Ansatz ist konzeptionell falsch und bietet keinerlei Sicherheitsvorteil: Annahme ist, dass der Angreifer zunächst einen Wert $salt'_{wrong}(uid, pw)$ bestimmt

und dazu eine Rainbow-Table aufbaut. Stattdessen ist zum Aufbau der Rainbow-Table jedoch für jeden Passwort-Kandidaten pw_x direkt folgendes berechenbar: $hash_{cli}^{slow}(kdf_{cli}(salt_{cli} \parallel pw_x, uid) \parallel pw_x, cost_{cli})$
Der Ansatz, pw in das Salting einzustreuen, ist nichtig, da eine Ableitung von zwei getrennten Faktoren aus pw ohne weitere Entropie-Quellen unmöglich ist.

Ein Angreifer könnte auch den Angriff auf pw'' zur Bestimmung von pw versuchen, siehe Kapitel 3.2.1. Dazu berechnet er Client-Hashwerte für typische Passwörter. Zu einem späterem Zeitpunkt gewinnt er Einblick in die DB und kennt damit pw''. Wenn er auch das Server-Pepper in Erfahrung bringt, kann er $hash_{svr}^{slow}$ auf die Kandidaten für pw' anwenden. Die Sicherheit der Passwortverarbeitung hängt damit weiterhin von der Ineffizienz des Server-Hashings, also von $cost_{svr}$, ab.
Zur Verdeutlichung sei ein Angreifer gegeben, der für einen Account per $OSINT$ mögliche Passwort-Bestandteile gesammelt hat. Daraus hat er ein Wörterbuch mit 10^{11} Einträgen erstellt. Er vermutet $|pw| = 8$ sowie $pw \in \Sigma^*$ mit Σ als Alphabet mit $|\Sigma| = 72$. Ein Passwort-Zeichen lässt sich damit in 7 Bit speichern. Der Angreifer verfügt über 10^3 parallele Prozessoren. Sei $t_{cli} = 100ms = 0,1s$ und $t_{svr} = 5ms = 0,005s$. Sei $hash_{cli}^{slow} = bcrypt$ mit $|bcrypt| = 184$ Bits.
Zur Vorbereitung des Angriffs muss der Angreifer Kandidaten für pw' bilden. Das erfordert einen Speicheraufwand von $10^{11} * (7 * 8 + 184) \approx 2,7\ TiB$. Die vorbereiteten Werte umfassen $10^{11}/72^8 \approx 0,014\%$ des Suchraums. Der Zeitaufwand zur Vorbereitung beträgt $10^{11} * 0,1s/10^3 \approx 116d$. Sobald der Angreifer pw'' kennt, versucht er pw zu bestimmen. Nach spätestens $10^{11} * 0,005s/10^3 \approx 5,8d$ hat er pw bestimmt oder weiß, dass der Wert nicht enthalten war.
Der IDP-Betreiber kann durch die Wertewahl für $cost_{cli}$ und $cost_{svr}$ den erwartbaren Zeitaufwand für Vorbereitung und Durchführung des Angriffs bestimmen. Folgende Strategien sind möglich:

- Der Dienstbetreiber wählt einen hohen Wert $cost_{svr}$, wie im Referenz-Protokoll. Der Aufwand zur Vorbereitung ist für einen Angreifer dann wenig effektiv. Allerdings ist auch die Serverlast höher, da das Client-Hashing nur unterstützend wirkt.
- Der Dienstbetreiber macht eine Angriffsvorbereitung so ineffektiv wie möglich. Dazu ist ein hoher Wert $cost_{cli}$ zu wählen. Der Wert $cost_{svr}$ und damit die Serverlast kann dann gering gehalten werden.

Der zweite Ansatz kann durch regelmäßige Salt-Änderung verbessert werden. Sofern der alte Wert pw'' nicht mehr bekannt wird, macht dies eine neue Angriffsvorbereitung nach jeder Salt-Änderung notwendig. Die Umsetzung dieser regelmäßigen Salt-Änderung wird nachfolgend erläutert.

Schutz durch Parameter-Rotation

Wie erläutert, kann das Client-Salt nicht geheim gehalten oder an das Passwort gekoppelt werden. Deshalb wird alternativ die einem Angreifer eingeräumte Zeitspanne zur Vorbereitung von Hashwerten gegen das Client-Hashing begrenzt. Dazu ist eine Rotation des Salts in regelmäßigen Abständen vorzunehmen, z. B. quartalsweise. Bei jeder Rotation wird ein neues Client-Salt $salt'_c$ gebildet, mit $c \in \mathbb{N}_0$ als fortlaufend inkrementiertem Zähler. Serverseitig kann zu jedem c ein $salt_c := kdf_{svr}(salt_{cli}, c)$ berechnet werden. Da für jede Rotation ein anderer Zählerwert vorliegt, ist die Nonce-Bedingung erfüllt. Der Server muss $salt_{cli}$ geheimhalten, um das Verfahren sicher zu gestalten. Falls ein Leak $salt_{cli}$ umfasst, muss der Wert geändert und der Zähler erhöht werden. Ausgestellte Client-Salts sind separat zu speichern, damit sie von einer Änderung des $salt_{cli}$ nicht betroffen sind. Die Bildung von Client-Salt und -Hashwert eines Nutzers für einen aktuellen Zählerwert c wird vorgenommen als:

$$salt'_{cli}(uid) := kdf_{cli}(salt_c, uid)$$
$$pw'_c := hash^{slow}_{cli}(kdf_{cli}(salt_c, uid) \parallel pw, cost_{cli})$$

Nach Durchführung einer Rotation liegt in der *DB* für einen davor registrierten Account pw''_{c-1} vor. Zur Aktualisierung des gespeicherten Wertes auf pw''_c muss der Nutzer sich anmelden und pw'_c vorlegen. Um den Nutzer dabei authentifizieren zu können, bedarf es einer Übergangszeit, in der $salt_c$ und $salt_{c-1}$ gültig sind. Initial sollte mit einem Zählerwert von $c = 1$ gestartet werden, damit sich stets ein Wert $c - 1$ bestimmen lässt.

Der Anmeldeprozess funktioniert wie folgt: Der Client erhält vom Server neben $cost_{cli}$ auch die Werte $salt_c$ und $salt_{c-1}$ zugesandt. Der Nutzer gibt *uid* und *pw* ein. Der Client berechnet pw'_c und pw'_{c-1} und übermittelt beide Werte an den Server. In der *DB* muss der für den Nutzer aktuelle Zählerwert c_{uid} hinterlegt sein. Zur weiteren Anmeldung agiert der Server wie im Referenz-Protokoll in Kapitel 2.3.2 beschrieben. Für einen existierenden, verifizierten Account werden zusätzlich die in Listing 3.1 dargestellten Aktionen ausgeführt.

```
1  if c_uid = c:
2      pw''_request := hash_password(pw'_c , uid, salt_db, pepper_svr, cost_db)
3      ok := (pw''_db = pw''_request)
4  else if c_uid =  c-1:
5      pw''_request := hash_password(pw'_c-1 , uid, salt_db, pepper_svr, cost_db)
6      ok := (pw''_db = pw''_request)
7  else:
8      pw''_request := hash_password(pw'_c , uid, salt_db, pepper_svr, cost_db)
9      ok := (pw''_db = pw''_request ∧ False)
10 if ok:
11     if c_uid = c-1:
```

```
12          pw''_new := hash_password(pw'_c , uid, salt_new, pepper_svr, cost_wanted)
13          update_account_record(pw''_new , cost_wanted)
14          increment_counter(c_uid)
15      create_session(uid)
16  else:
17      error: uid or pw wrong or account expired
```

Listing 3.1: Serverseitige Anweisungssequenz zur Anmeldung mit Salt-Rotation

Der Server überprüft zunächst, ob der Account über einen akzeptablen Zählerwert verfügt. Andernfalls ist der Account abgelaufen und ein Login via Passwort nicht möglich. Zur Gewährleistung einer zeitkonstanten Verarbeitung werden in diesem Fall trotzdem die Operationen ausgeführt (Zeile 8-9), siehe Kapitel 2.3.2. Sind die Zugangsdaten korrekt und der Account aktuell, muss der Hashwert zum aktuellen Zählerwert gespeichert werden, falls noch der zum vorhergehenden hinterlegt war (Zeile 11-14). Dabei werden auch ein neues Server-Salt und der aktuelle Kostenwert verwendet, siehe Kapitel 2.3.5. Ein weiterer Vorteil des Verfahrens besteht darin, dass bei jeder Rotation die Entitäten $cost_{cli}$ und $cost_{svr}$ eines Accounts aktualisiert werden können. Deshalb sollte für das Client-Hashing immer eine Rotationsmöglichkeit vorgesehen werden.

Während die Nachteile des Verfahrens im Backend gering sind, ergeben sich im Frontend größere Belastungen: In der Übergangszeit nach einer Rotation sind für jeden Login zwei aufwändige Hash-Vorgänge auszuführen, da dem Frontend nicht bekannt ist, ob der Hashwert in der *DB* schon aktualisiert wurde. Die Mitteilung dieser Angabe durch das Backend würde eine Informationspreisgabe darstellen, welche Diskussionsthema von Kapitel 3.2.3 ist. Stattdessen könnte das Frontend auch zuerst eine Anmeldung mit pw'_c versuchen und im Fehlerfall mit pw'_{c-1}. Das würde bis zu zwei *Roundtrips* erfordern. Abhängig von Netzwerklatenz und Hashing-Geschwindigkeit des Endgeräts kann der *UA* die geeignetere der beiden Strategien adaptieren.
Weiteres Optimierungspotenzial für durch Client-Hashing bedingte Wartezeiten bietet *Just In Time (JIT)*-Hashing. Hierbei finden erste Hashberechnungen bereits während der Passworteingabe statt, sodass der Nutzer sie weniger als Wartezeit wahrnimmt. Da etablierte Hash-Algorithmen in der Regel keinen *JIT*-Modus vorsehen, sind bestehende Implementierungen zu modifizieren.[HB18] Allerdings kann dieser Eingriff neue Designprobleme und kritische Bugs verursachen.

Das Rotationsverfahren verschlechtert die *UX*: Nimmt ein Nutzer innerhalb der Übergangszeit nach einer Rotation keine Änderung seines Passwortes vor, wird das Passwort als Authentifizierungsmerkmal für ungültig erklärt. Der Account ist dann als deaktiviert zu betrachten. Um wieder Zugriff zu erlangen, muss der betroffene Anwender eine Zurücksetzung der Anmeldedaten vornehmen, siehe Ka-

pitel 2.3.4. Zur Vermeidung dieser Prozedur können Gegenmaßnahmen ergriffen werden: Die Übergangszeit sollte groß gewählt sein, bis maximal zur Lebenszeit einer Rotation. Nach Durchführung einer Rotation sollten alle verifizierten Nutzer per Mail zur einmaligen Anmeldung aufgefordert werden, um den Aktiv-Status des Accounts beizubehalten. Das Einfordern dieses Zwangs-Logins verschlechtert jedoch die *UX* und kann abhängig von Benutzerkreis und Anwendungstyp als Gängelung oder Überwachung wahrgenommen werden.

Das Verfahren besitzt eine Schwäche bezüglich des Umgangs mit historischen *DB*-Beständen: Diese können Angreifer z. B. unzureichend gesicherten Cloud-Backups entnehmen. Wenn der Angreifer auch die Server-Pepper in Erfahrung bringt, sind Angriffsvorbereitungen vergangener Rotationen dennoch effizient und zielführend anwendbar. Das Backup der Accountdaten vom restlichen *DB*-Backup zu separieren, verkompliziert die Backup-Logistik und ist anfällig für organisatorische Fehler. Statt des Zwangs-Logins eine regelmäßige Passwortänderung einzufordern, birgt die in Kapitel 2.3.5 erläuterten Probleme zur Wahl ähnlicher Passwörter. Daher besteht der beste Schutz vor dem Angriff mit vorberechneten Hashwerten in einer peniblen Geheimhaltung der Server-Pepper.

Zusammengefasst kann Client-Salting den Schutz vor Angriffen mit vorausberechneten Hashwerten stärken, insbesondere wenn eine Salt-Rotation implementiert wird. Allerdings verschlechtert Salt-Rotation die *UX* bei begrenzter Schutzwirkung. Eine Rotationsmöglichkeit für Client-Parameter sollte dennoch bedacht und etabliert werden, um $cost_{cli}$ unkompliziert erhöhen zu können. Hierfür genügen größere Rotations-Intervalle, z. B. zweijährlich. Eine mögliche Alternative zur Salt-Rotation besteht im nachfolgend erläuterten Client-Peppering.

3.2.3 Client-Hashing mit Salt und Pepper

Das Client-Peppering ergänzt das Client-Salting. Das Pepper muss einer Geheimhaltung unterliegen, um eine sinnvolle Erweiterung zum Salting zu bieten. Der in Kapitel 3.2.2 vorgestellte Ansatz zum Client-Salting ist dafür nicht geeignet. Eine clientseitige Pepper-Konstruktion unter Einbeziehung des Passwortes ist ebenfalls nicht möglich, siehe Kapitel 3.2.2. Stattdessen muss das Client-Pepper als Secret für jeden Nutzer individuell vorgehalten werden. Da es jeder *UA* zum Login benötigt, müssen Client-Pepper serverseitig gespeichert werden.[1] Ausgehend von Kapitel 3.2.2 wird angenommen, dass die Festlegung der Client-Salts statisch (ohne Rotation) erfolgt, weshalb die Pepper bei einem Passwortwechsel zu ändern sind. Nachfolgend werden vier aufeinander aufbauende Ansätze diskutiert.

[1] Das Server-Hashing ist unabhängig vom clientseitigen und nutzt andere Salts sowie Peppers.

Schema 1: Client-generierte Client-Pepper

Zur Registrierung wählt der Client mittels *CSPRNG* ein $pepper_{cli}$ und übermittelt es mit der regulären Registrierungsanfrage an den Server.
Der Anmeldeprozess aus Kapitel 2.3.2 wird um einen Anfangsschritt erweitert: Der Client bezieht nach Eingabe der *uid* das zugehörige $pepper_{cli}$ vom Server. Unter Einbeziehung des Salting-Verfahren aus Kapitel 3.2.2 erfolgt die Hashberechnung als:

$$pw' := hash_{cli}^{slow}(pepper_{cli} \parallel salt'_{cli}(uid) \parallel pw, cost_{cli})$$

Das Problem dieses Protokolls besteht in der Informationspreisgabe der Client-Peppers an nicht-authentifizierte Aufrufer. Die Pepper sind damit als öffentlich bekannt zu betrachten. Sie fungieren lediglich als zweites Salt. Problematisch ist die Angriffsmöglichkeit der User-Enumeration: Ein Angreifer kann sich für beliebige Nutzerkennungen das Client-Pepper zurückgeben lassen. Für nicht-existente Accounts ist eine Sonderbehandlung zu treffen. Die Rückgabe eines Zufallswerts ist nicht möglich, da Angreifer durch doppelte Anfragen die Abweichung feststellen würden. Für jeden angefragten, nicht-existenten Account ein Pepper zu generieren und vorzuhalten, würde aufgrund des steigenden Speicherbedarfs einen *DOS*-Angriff ermöglichen. Stattdessen kann serverseitig ein zentraler Wert $pepper_{\varnothing}$ hinterlegt werden, der geheim zu halten ist. Aus diesem werden die Pepper für nicht-existente Accounts abgeleitet: $pepper'_{cli}(uid) := kdf_{svr}(pepper_{\varnothing}, uid)$.

Allerdings ist diese Lösung unvollständig, da sich das Pepper für nicht-existente Accounts nie ändert. Im Fall von Registrierung oder Passwortänderung sollte auch das Client-Pepper geändert werden. Andernfalls würde durch die Maßnahme kein Vorteil gegenüber den Client-Salts entstehen, siehe Kapitel 3.2.2.
Eine regelmäßige Änderung, z. B. jährlich, von $pepper_{\varnothing}$ löst das Problem auch nicht komplett, da sich dann sämtliche daraus erzeugten Client-Pepper zeitgleich ändern. Ein Angreifer hätte damit umso mehr die Bestätigung, dass ein Account nicht existiert. Einziger Vorteil wäre die erhöhte Wartezeit zur Feststellung. Diese Verschleierungstechniken sind noch weiter verfeinerbar. Letztendlich kann ein Angreifer aber immer das Verhalten von Client-Peppern zu Nutzerkennungen, die sehr wahrscheinlich nicht existieren, denen gegenüber stellen, die wahrscheinlich existieren. Zudem ist fraglich, ob ein Nutzer sein Account-Passwort überhaupt ändert, wenn er nicht dazu gezwungen wird, siehe Kapitel 2.3.5.[ith15]

Neben der Exponiertheit der Existenz von Accounts besteht ein weiterer Protokoll-Fehler in der Privatsphäre-Verletzung der Anwender. Sei beispielsweise eine Dating-App gegeben, bei der das Opfer einen Account besitzt. Einem Stalker

sei die Nutzerkennung bekannt. Eine Änderung des Client-Peppers nach langer Zeit könnte aus einer Zurücksetzung der Anmeldedaten resultieren und anzeigen, dass der Nutzer die App wieder verwendet. In diesem Szenario würden Einblicke in das Privatleben des Opfers ermöglicht. In anderen Konstellationen muss die Informationspreisgabe kein Ausschlusskriterium für das Verfahren sein. Jedoch überwiegen die Probleme den Nutzen, zumal die Pepper nicht geheim sind.

Schema 2: Server-generierte Client-Pepper

Dieser Ansatz erzeugt die Pepper nicht mittels *CSPRNG* sondern einer *KDF*. Dazu wird zu jedem Account in der *DB* ein Zählerwert $ctr \in \mathbb{N}_0$ festgehalten, welcher die Anzahl an vollführten Passwortänderungen angibt. Für den Anmeldungsprozess übermittelt der Client zunächst die betreffende *uid*. Der Server schlägt den zugehörigen Wert *ctr* in der *DB* nach. Dem Client sendet er daraufhin das Client-Pepper, was er wie folgt erzeugt: $pepper_{cli} = kdf(pepper_{users}, uid \parallel ctr)$.
Der Server verfügt zur Konstruktion über einen geheimen Wert $pepper_{users}$. Es gilt den Sonderfall nicht-existenter Accounts zu betrachten. Anders als beim vorherigen Ansatz soll das Nutzerverhalten zur Passwort-Änderung nicht nachgeahmt werden. Stattdessen wird *ctr* für nicht-existente aus existenten Accounts abgeleitet. Der Server muss dafür eine von außen nicht nachvollziehbare, injektive Zuordnung zwischen nicht-existenten und existenten Nutzerkennungen treffen. Voraussetzung für die Intransparenz ist, dass ein einzelner Angreifer nicht über den Großteil aller Accounts verfügen darf. Folgender Ansatz wird vorgeschlagen: Für eine angefragte, nicht-existente Nutzerkennung uid_{req} wird unter Einsatz einer weiteren *KDF* berechnet: $idx := kdf_{idx}(pepper_{users}, uid_{req})$
Der Wert *idx* wird als vorzeichenlose Ganzzahl interpretiert. Er gibt den *DB*-Index (Zeilennummer) des zugehörigen existierenden Accounts an. Dazu muss bei n Accounts $n \geq 2^{|kdf_{idx}|}$ gelten. Problematisch ist der Umgang mit einer steigenden Nutzeranzahl: Wenn $|kdf_{idx}|$ an n angenähert wird, ändert sich die Zuordnung, sobald die Nutzerzahl sich verdoppelt hat. Damit funktioniert das Verfahren nur für Anwendungen mit großer Account-Anzahl stabil. Falls ein Nutzerstamm erst aufzubauen ist, können initial künstliche, nicht-verifizierte Accounts mit zufälligen Zählerwerten angelegt werden. Die generierten Nutzerkennungen sollten so gewählt werden, dass sie zukünftig wahrscheinlich durch Menschen registriert werden. Wenn z. B. rund eine Million Nutzer erwartet werden, können initial $2^{20} = 1.048.576$ Accounts erstellt werden.
Der Registrierungsprozess aus Kapitel 2.3.1 ist mit dem Ansatz nicht kompatibel, da zur Registrierung das Pepper mit $ctr = 0$ benötigt wird. Deshalb wird der Prozess umgekehrt: Zunächst findet eine Mail-Verifikation und danach eine

mittels Bearer-Token authentifizierte Passwortvergabe statt. Durch diese Authentifizierung wird der Nutzer berechtigt, das echte Client-Pepper (mit $ctr = 0$) zum Account zu beziehen.

Schema 3: Server-generierte, anfrage-abhängige Client-Pepper

Die *KDF*-Konstruktion der Pepper löst noch nicht das Geheimhaltungsproblem. Dazu ist die Server-Interaktion zur Pepper-Herausgabe von einem Anfragedatum abhängig zu machen. Folgender Ansatz wird vorgeschlagen: Der Client bildet nach dem Prinzip der k-Anonymität aus Kapitel 3.1 das Präfix vom Passwort-Hashwert mit a Bits: $prefix := hash_{cli}^{slow}(salt'_{cli}(uid) \parallel pw, cost_{prefix})[0..a]$
An den Server werden $(uid, prefix)$ übertragen, welcher die Werte zur Pepper-Generierung nutzt: $pepper_{cli} = kdf(pepper_{users}, uid \parallel ctr \parallel prefix)$
Um den gesamten Pepper-Suchraum eines Accounts herunterzuladen, muss ein Angreifer 2^a Anfragen stellen. Ein legitimer Nutzer muss nur eine oder wenige Pepper-Anfragen stellen, sodass ein starkes Rate-Limiting implementierbar ist, siehe Kapitel 5.1.5. In diesem besteht die Schutzwirkung des Verfahrens. Sobald ein verdächtiger Client identifiziert wurde, kann ein abweichender Wert $pepper_{limit}$ anstatt $pepper_{users}$ zur Pepper-Bildung genutzt werden. Diese Abweichung ist für den Client nur durch wiederholte Anfragen feststellbar, was die Wahrscheinlichkeit zur Anwendung des Rate-Limitings erhöht.
Der Parameter a befindet sich in einem Spannungsfeld: Je größer er ist, umso mehr Präfix-Anfragen muss ein Angreifer stellen. Bei z. B. $a = 24$ Bits existieren $2^{24} = 16.777.216$ Möglichkeiten. Andererseits erlaubt ein abgefangenes Präfix eine feinere Vorfilterung potenzieller Passwort-Kandidaten. Deshalb ist $cost_{prefix}$ nicht zu niedrig (aber kleiner als $cost_{cli}$) zu wählen. Damit muss der Client wie bei der Salt-Rotation zwei aufwändige Hash-Operationen ausführen, um das Präfix und den eigentlichen Passwort-Hash zu bestimmen. Würde $cost_{cli} = cost_{prefix}$ gewählt, so könnte der Angreifer mit nur einer anstatt zwei Hashtables vom Präfix auf mögliche Passwörter schließen. Um darüber hinaus eine mögliche Kombination der Hashing-Vorgänge zum Aufbau beider Hashtables ausschließen zu können, sind zwei verschiedene, langsame Hashfunktionen einsetzbar.
Vorteilhaft am Verfahren ist, dass im Backend zu keinem Zeitpunkt Client-Pepper persistiert werden. Aus einem *DB*-Leak inklusive $pepper_{users}$ ergeben sich 2^a mögliche Client-Pepper je Account. Die gewünschte Geheimhaltung der Pepper ist damit bei hinreichend großem a gegeben. Allerdings wird das Privatsphäre-Problem nicht gelöst: Ein Angreifer kann regelmäßig ein Präfix für einen Account anfragen und die Gleichheit der Antworten prüfen, um eine Zählererhöhung festzustellen. Dies wird nachfolgend behoben.

Schema 4: Server-generierte, authentifizierte Client-Pepper

Das anfrage-abhängige Client-Peppering wird um eine Authentifizierung ergänzt. Der Server benötigt dafür ein Vergleichsmerkmal. Dazu ist das besagte Präfix nutzbar, welches als $prefix_{db}$ zu einem Account in der *DB* gespeichert wird. Auf eine Client-Anfrage mit den Parametern *uid* und $prefix_{req}$ führt der Server die in Listing 3.2 gezeigten Schritte aus und sendet das Ergebnis $pepper_{cli}$ zurück. Die Werte $pepper_{users}$ und $pepper_{fake}$ sind verschiedene Server-Secrets.

```
1 idx ≔ kdf_idx(pepper_fake, uid_req‖prefix_req) as unsigned integer
2 ctr_user ≔ query_pw_change_counter_by_uid_from_db(uid)
3 ctr_fake ≔ query_pw_change_counter_by_account_index_from_db(idx)
4 if prefix_req = prefix_db:
5     pepper_cli ≔ kdf(pepper_users, uid‖ctr_user)
6 else:
7     pepper_cli ≔ kdf(pepper_fake, uid‖ctr_fake)
```

Listing 3.2: Serverseitige Anweisungssequenz zur zeitsicheren Konstruktion authentifizierter Client-Pepper

Der Code wurde zeitsicher entworfen, weshalb der Overhead einer unnötig ausgeführten *DB*-Abfrage je Anfrage entsteht (Zeile 2-3). Ein Angreifer kann durch Variation von $prefix_{req}$ eine Chosen-Plaintext-Attacke ausführen. Jedoch ist er nicht in der Lage, das richtige Präfix aus der Server-Antwort zu bestimmen, da jedes zurückgegebene *Opaque* ein valides Pepper darstellt. Durch die indexbasierte Zuordnung falscher Präfixe zu anderen Accounts wird die Privatsphäre des Nutzers geschützt. Wie erläutert, bedarf das Verfahren hierzu einer hinreichenden Poolgröße an Accounts. Unter der Bedingung $pepper_{users} \neq pepper_{fake}$ sind gesammelte Antwort-Pepper nicht für zukünftige Zählerstände nutzbar.
Neben den genannten Vorteilen entstehen auch Sicherheitsprobleme: Da $prefix_{db}$ zum Account in der *DB* gespeichert ist, kann ein Angreifer den Wert im Falle eines *DB*-Leaks verwenden. Solange er das außerhalb der *DB* aufbewahrte $pepper_{users}$ nicht kennt, ist keine aufwandsarme Bestimmung des Client-Peppers möglich. Allerdings ist es als Filter zur Dictionary-Attacke verwendbar. Deshalb muss das clientseitig genutzte $cost_{prefix}$, wie oben erläutert, hoch sein. Ein Hashing von $prefix_{db}$ zur Speicherung ist aufgrund des kleinen Suchraums nicht sinnvoll.

Durch die erläuterten Maßnahmen konnten die am Anfang des Abschnitts erläuterten Probleme des Protokolls mit Client-Peppering ausgeräumt werden. Die Risikobetrachtung zur Bewertung der getroffenen Maßnahmen folgt.

3.2.4 Risiko-Bewertung

Nachfolgend wird eine Risiko-Bewertung zum Client-Hashing vorgenommen, wobei die Protokollvarianten bezüglich Salting und Peppering unterschieden werden.

Die Risiko-Bewertung basiert auf der Angreifer-Klassifikation aus Kapitel 2.1.2 und nutzt das Threat-Modelling-Verfahren DREAD-D, welches in Kapitel 2.1.3 erläutert wurde. Zur differenzierten Risiko-Bewertung sind die in Tabelle 3.11 dargestellten Protokoll-Varianten CP1 bis CP4 zu unterscheiden.

Tabelle 3.11: Protokollvarianten zur Risiko-Betrachtung von clientseitigem Hashing

Protokoll-Variante	Client-Salting	Rotation der Client-Paramter	Client-Peppering	Pepper-Verwahrung	Referenz
CP1	×				Kapitel 3.2.2
CP2	×	×			Kapitel 3.2.2
CP3	×		×	anonym	Kapitel 3.2.3, Schema 3
CP4	×		×	in der DB	Kapitel 3.2.3, Schema 4

Risiko-Bewertung zu CP1

Mit dauerhaft statischen Client-Salts besteht das Problem der Vorbereitungsmöglichkeiten durch den Angreifer, siehe Tabelle 3.12. Das Schema ist damit als Unterstützung eines aufwändigen Server-Hashings geeignet. Darüber hinaus kann es für ein Hashing mit *Server-Relief* genutzt werden, wenn die Accounts nur über eine zeitlich begrenzte Lebenszeit verfügen, z. B. beim Einsatz im Rahmen einer Schulung oder Konferenz. In diesem Fall wird, wie in Tabelle 3.8 gezeigt, eine Server-Überlastung bei vielen parallelen Logins verhindert. Das Schema bleibt jedoch nur bedingt für Applikationen mit langfristig schützenswerten Accounts geeignet. Die Gefahr einer Einblicknahme der Accountdaten sowie Server-Pepper am Backend durch Dritte ist von den getroffenen technischen und organisatorischen Maßnahmen der konkreten Anwendung abhängig und daher dem Threat-Modelling an dieser Stelle nicht zugänglich. Jedoch wird im Weiteren davon ausgegangen, dass dies nur mit erhöhtem Aufwand und technischer Expertise möglich ist, vergleiche Tabelle 3.2 in Kapitel 3.1.3.

Tabelle 3.12: Risiko für CP1 zum Angriff mit vorberechneten Hashwerten

D	4/5	Grundlage für Angriff gegen das begrenzt aufwändige Server-Hashing
R	2/5	Hoher, aber parallelisierbarer Ressourceneinsatz notwendig
E	1/5	Ausnutzung bedarf eines Zugriffs auf DB und Server-Pepper
A	3/5	Betrifft jeweils nur einen Account, aber Angriff parallelisierbar
$\sum$	10/20	$\Rightarrow$ **Mittleres Risiko**

Risiko-Bewertung zu CP2

Durch Rotation der Parameter zum Client-Hashing muss der Angreifer nicht nur einen hohen Vorbereitungsaufwand betreiben, sondern dies auch in begrenzter Zeit leisten können, siehe Tabelle 3.13. Weiterhin dürfen historische Passwort-Hashes in *DB*-Dumps nicht zusammen mit dem Server-Pepper bekannt werden. Hierzu ist die Risiko-Einschätzung aus Tabelle 3.2 in Kapitel 3.1.3 übertragbar.

Wenn diese Geheimhaltung gewährleistet wird, ist das Verfahren sowohl zur Unterstützung eines aufwändigen Server-Hashings als auch zum Server-Relief-Hashing geeignet. Falls die Rotationen nicht in regelmäßigen Intervallen erfolgen, so ist für einen starken Schutz eine verpflichtende *MFA* zu ergänzen, da sonst wieder die Risiko-Betrachtung zu CP1 gilt.

Tabelle 3.13: Risiko für CP2 zum Angriff mit vorberechneten Hashwerten

D	4/5	Grundlage für Angriff gegen das begrenzt aufwändige Server-Hashing
R	1/5	Hoher, parallelisierbarer Ressourceneinsatz in begrenzter Zeit notwendig
E	1/5	Ausnutzung bedarf eines Zugriffs auf DB und Server-Pepper
A	3/5	Betrifft jeweils nur einen Account, aber Angriff parallelisierbar
$\sum$	9/20	$\Rightarrow$ **Niedriges Risiko**

Risiko-Bewertung zu CP3

Im Gegensatz zum Client-Salting bedarf das Client-Peppering einer Interaktion mit dem Server, um den Parameterwert zu bestimmen. Damit wird der Angriffsvektor auf das Backend ausgeweitet. Wie in Tabelle 3.14 dargestellt, wird damit ein *DDOS*-Angriff ermöglicht. Aufgrund des Rate-Limitings und der nur schwachen erzeugbaren Server-Last ist dieser jedoch nicht bedrohlicher als andere Ansatzpunkte zum *DDOS*, vergleiche Kapitel 3.1.3.

Ein Angriff mit vorberechneten Hashwerten bleibt weiter möglich, wobei jedoch für jedes mögliche Pepper separate Vorbereitungen zu treffen sind. Dazu muss der Angreifer zunächst sämtliche Client-Pepper eines Nutzers bei der Pepper-Ausgabe des Servers erfragen. Durch das Rate-Limiting wird dies ausgebremst, wobei mit hinreichend hoher Laufzeit auch das Rate-Limiting umgehbar ist. Wie Tabelle 3.15 entnommen werden kann, wird der Angriff durch Wahl einer hinreichenden Pepper-Suchraumgröße entschärft. Voraussetzung bleibt, dass der Angreifer das richtige Client-Pepper nicht kennt. Andernfalls ist ein Angriff mit vorberechneten Hashwerten wie zu CP1 möglich.

Tabelle 3.14: Risiko für CP3 durch DDOS-Attacke gegen die Pepper-Ausgabe

D	2/5	Eingeschränkte Verfügbarkeit solange der Angriff anhält
R	3/5	Angriff benötigt hohen aber verteilten Ressourceneinsatz
E	2/5	Umgehung Rate-Limiting zur Ausnutzung notwendig
A	4/5	Ausfall des Backends betrifft potenziell viele Nutzer
$\sum$	**11/20**	$\Rightarrow$ **Mittleres Risiko**

Tabelle 3.15: Risiko für CP3 zum Angriff mit vorberechneten Hashwerten

D	3/5	Grundlage für Angriff gegen Server-Hashing
R	1/5	Aufwändiges Hashing für jedes mögliche Pepper
E	2/5	Abfrage sämtlicher Pepper, Umgehung Rate-Limiting
A	3/5	Jeweils ein Account betroffen, Angriff parallelisierbar
$\sum$	**9/20**	$\Rightarrow$ **Niedriges Risiko**

CP3 besitzt das gravierende Privatsphäre-Problem des *Account-Stalkings* bezüglich des Passwortwechsels, siehe Tabelle 3.16.
Eine User-Enumeration wird durch CP3 verhindert. Allerdings kann ein Angreifer sich darüber hinwegsetzen und die Zuordnung zwischen Accounts transparent machen. Dazu legt er Accounts an, bis ihm mehr als die Hälfte gehören. Das Risiko dieser Account-Schwemme wird in Tabelle 3.17 dargestellt.

Tabelle 3.16: Risiko für CP3 zum Account-Stalking

D	2/5	Privatsphäre-Verletzung, Beobachtung Passwortwechsel
R	5/5	Skript-basierte Überwachung möglich
E	5/5	Betrachtung Client-Quellcode ausreichend
A	4/5	Angriff parallelisierbar, Rate-Limiting irrelevant
$\sum$	**17/20**	$\Rightarrow$ **Hohes Risiko**

Tabelle 3.17: Risiko für CP3 durch Account-Schwemme

D	2/5	User-Enumeration, Beobachtung Passwortwechsel
R	2/5	Automatisierte Account-Erstellung notwendig
E	3/5	Umgehung Rate-Limiting zur Ausnutzung notwendig
A	5/5	Angriff betrifft sämtliche Accounts
$\sum$	**12/20**	$\Rightarrow$ **Mittleres Risiko**

Alternativ kann ein Angreifer zur User-Enumeration eine Timing-Attacke starten, was allerdings durch das Protokoll-Design mitigiert wird, siehe Tabelle 3.18.
Als *MITM* könnte ein Angreifer die Übermittlung von Hash-Präfixen zur Pepper-Bildung aufzeichnen, siehe Tabelle 3.19. Auf die Präfix-Übermittlung folgt als lohnenderes Ziel jedoch stets die Übertragung des Passwort-Hashwertes, welche in Tabelle 3.3 in Kapitel 3.1.3 evaluiert wurde. In der Gesamtbetrachtung steigert eine Präfix-Übertragung das *MITM*-Risiko damit nicht.

Tabelle 3.18: Risiko für CP3 zur Timing-Attacke gegen die Pepper-Ausgabe

D	2/5	Durchführung einer User-Enumeration
R	1/5	Präzise, auswertbare Zeitmessung
E	3/5	Viele wiederholte Anfragen trotz Rate-Limitings
A	2/5	Betrifft jeweils einen Account, Angriff parallelisierbar
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Tabelle 3.19: Risiko für CP3 zum MITM gegen die Pepper-Übertragung

D	4/5	Einschränkung des Passwort-Suchraums
R	2/5	Abfangen übertragener Werte und aufwändiges Hashing nötig
E	1/5	Einbruch in TLS-Verbindung notwendig
A	1/5	Nur Nutzer kompromittierter Verbindungen betroffen
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Risiko-Bewertung zu CP4

CP4 mitigiert das *Account-Stalking*. Allerdings können bei Versagen der installierten Mitigationen eine User-Enumeration durchgeführt, ein Passwortwechsel beobachtet oder auch das korrekte Präfix und damit Details des Schlüsselmaterials extrahiert werden. Via Account-Schwemme ist eine User-Enumeration und infolgedessen *Account-Stalking* nach wie vor möglich, siehe Tabelle 3.17. Ansonsten erfordert der Angriff die Kenntnis des richtigen Präfixes.

CP4 führt gegenüber CP3 eine Authentifizierung für Client-Pepper mit Klartext-Speicherung der Präfixe ein. Ein Angreifer kann aus einem *DB*-Auszug die Präfixe bestimmen und damit die Pepper am Server erfragen, siehe Tabelle 3.20. Alternativ zur Account-Schwemme kann ein Angreifer zur User-Enumeration weiterhin eine Timing-Attacke gegen die Pepper-Ausgabe richten. Wenn diese Erfolg hat, kann der Angreifer auch das korrekte Präfix bestimmen, siehe Tabelle 3.21.

Tabelle 3.20: Risiko für CP4 zur Timing-Attacke gegen die Pepper-Ausgabe

D	4/5	Präfix-Bestimmung, Einschränkung des Suchraums mit vorberechneten Hashwerten
R	1/5	Präzise, auswertbare Zeitmessung
E	3/5	Viele wiederholte Anfragen trotz Rate-Limitings
A	2/5	Betrifft jeweils einen Account, Angriff parallelisierbar
$\sum$	**10/20**	$\Rightarrow$ **Mittleres Risiko**

Tabelle 3.21: Risiko für CP4 durch Angreifer mit Einblick in die DB

D	5/5	Dictionary Attacke inklusive Suchraum-Einschränkung mit vorberechneten Präfix-Hashes
R	2/5	Aufwändiges Hashing für Dictionary Attacke nötig
E	2/5	Zugang zu einem Datenauszug aus der DB
A	5/5	Angriff gegen sämtliche Accounts möglich
$\sum$	**14/20**	$\Rightarrow$ **Mittleres Risiko**

Beurteilung

Aus der Risiko-Bewertung folgt, dass für clientseitiges Hashing ein Salting inklusive optionaler Parameter-Rotation nach CP2 obligatorisch ist. Beim Server-Relief-Hashing ergibt sich das Schutzniveau aus dem Aufwand eines Angreifers zur Vorbereitung von Hashwerten sowie dem Zugang zu *DB*-Inhalten und Server-Pepper. Falls der Schutzbedarf eines Accounts derartige Vorbereitungen durch Angreifer plausibel oder wahrscheinlich erscheinen lässt, so ist die Ergänzung einer *MFA* geboten, siehe Kapitel 4.

Im Gegensatz zum Client-Salting eröffnet das Client-Peppering einen invasiven Angriffsvektor gegenüber dem Server. Gegen diese Lücken wurden Mitigationen installiert, wobei keine der Techniken ohne Restrisiko oder Einschränkungen funktioniert. Da die Mitigationen auf dem Zusammenspiel mehrerer Maßnahmen beruhen, sind sie potenziell fehleranfällig. Das Threat Modelling zeigt die große Angriffsoberfläche von CP3 und CP4, wobei an mehreren Stellen mittlere bis hohe Risiken festgestellt wurden.
Deshalb wird eine klare Empfehlung gegen den generellen Einsatz von Client-Peppering ausgesprochen.

Client-Peppering ist ausschließlich in Szenarien sinnvoll, in denen die Mitigationen verzichtbar sind. Dann wäre zur Stärkung der Protokoll-Sicherheit eine abgewandelte Form von CP3 mit folgenden Eigenschaften einsetzbar:

- Client-Pepper werden entsprechend CP3 gebildet und niemals persistiert, sodass sie im Falle eines *DB*-Leaks unbekannt bleiben.
- Keine Mitigationen für User-Enumerations: Keine Zuordnung anderer Account-Daten für nicht-existente Accounts.
- Keine Mitigationen für *Account-Stalking*: Beobachtung von Passwortwechseln durch Angreifer wird legitimiert.

Beispielsweise wäre ein valides Szenario für Client-Peppering der Einsatz im Backend zur Authentifizierung des Applikationsservers gegenüber der *DB*, da hier nur technische User in einem geschützten Bereich eingesetzt werden. Dann gilt es allerdings zu hinterfragen, warum für diese *M2M*-Kommunikation Passwörter eingesetzt werden und nicht eine stärkere, zertifikatsbasierte Authentifizierung, siehe Kapitel 4.3.2.

Aufgrund der begrenzten Einsatz-Szenarien wird im Folgenden von Client-Hashing mit Salting und ohne Peppering ausgegangen. Jedoch wäre das Peppering für alle nachfolgenden Protokolle problemlos ergänzbar, sofern die erläuterten Implikationen in Kauf genommen werden.

3.3 Challenge-Response-Verfahren zur Authentifizierung

Ein *Challenge Response Authentication Mechanism (CRAM)* ist ein Protokoll zur Authentifizierung. Die Partei des Verifiers gibt Challenges aus. Möchte ein Prover als zweite Partei sich authentifizieren, so erfragt er beim Verifier eine Challenge. Anschließend muss der Prover die korrekte Response auf die durch den Verifier gestellte Challenge an ihn zurücksenden. Die Identität des Provers ist gegenüber dem Verifier durch die Fähigkeit, die richtige Antwort zu erzeugen, nachgewiesen. In bisher vorgestellten Protokollen agierten Clients als Prover und der Server als Verifier. Eine wechselseitige Authentifizierung wird ermöglicht, indem Server und Client sich ihre Identität gegenseitig mittels *CRAM* bestätigen.

Im Folgenden werden eine Klassifikation der Verfahren vorgenommen und anhand von Beispielen mögliche Schwachstellen erläutert. Ausgehend vom Protokoll mit Client-Salting aus Kapitel 3.2.2 wird ein *CRAM*-Verfahren mit clientseitiger Passwortverarbeitung und *Server-Relief* konstruiert.

3.3.1 Klassifikation

In Kapitel 2.3.1 wurde im Zuge der Mail-Verifikation des Referenz-Protokolls bereits eine tokenbasierte Authentifizierung als *CRAM* verwendet: Der Nutzer erhält die Challenge in Form des signierten Verifikations-Tokens an sein Mail-Konto und muss ihn als Response beim Server vorlegen. Dadurch wird der Zugriff auf das Mail-Konto nachgewiesen.
In diesem Protokoll darf die Challenge der Response entsprechen, da lediglich der Zugriff zum Kommunikationskanal und nicht die Kenntnis eines geheimzuhaltenden Merkmals nachgewiesen wird. Da der Server vorgelegte Verifikations-Tokens invalidiert, ist eine Replay-Attacke nicht zielführend, zumal über den Bearer-Token nur eine einmalige Account-Verifikation erzielbar ist.

Die Mail-Verifikation ist jedoch ein Spezialfall eines *CRAM*: In der Regel vergibt der Verifier an anfragende Clients ohne Authentifizierung Challenges. Damit dürfen die Responses an dieser Stelle nicht den Challenges entsprechen. Stattdessen muss auf Basis der vorliegende Challenge eine Response durch den Prover bestimmt werden. Die Protokoll-Familie der *CRAM* wird hierbei ausgehend von den ausgestellten Challenges in statische und dynamische Verfahren unterteilt.

Statische Verfahren Für ein statisches Verfahren wird bei jeder Authentifizierungsanfrage eine Challenge aus einem begrenzten Pool an Challenge-Response-Tupeln gewählt. Spätestens wenn die Kapazität des Pools erschöpft ist, müssen

Challenges wiederverwendet werden. Beispielsweise verwenden Sicherheitsfragen zum Schutz des Zurücksetzens von Anmeldedaten diesen Ansatz, siehe Kapitel 2.3.4. Ein *Eavesdropper*, der ein Tupel aus Challenge und Response mitgeschnitten hat, kann sich damit authentifizieren, sobald die passende Challenge erneut gestellt wird.

Dynamische Verfahren Im Gegensatz dazu verwenden dynamische Verfahren keinen vorgegebenen Pool, sondern erzeugen für jede Anfrage eine neue Challenge, z. B. mittels *CSPRNG* als Zufallszahl. Die Wahrscheinlichkeit einer Kollision ergibt sich aus der Anzahl möglicher Challenges. Bei einer Challenge-Größe von zwei Byte sind dies lediglich $2^{16} = 65\,536$ Möglichkeiten. Für einen Angreifer kann es damit weiterhin lukrativ sein, Challenge-Response-Paare zu sammeln und Anfragen gegen den Verifier zu stellen, bis eine bekannte Challenge abgefragt wird. Dagegen kann die Challenge-Größe erhöht werden: Bei z. B. 16 Bytes ergeben sich $2^{128} \approx 3,4 * 10^{38}$ Möglichkeiten.
Wenn die Challenge-Größe durch das System limitiert ist, muss der Verifier kürzere Challenges vergeben. Um hierbei eine Kollision ausschließen zu können, sind die Challenges als Nonce zu behandeln. Hierzu ist beispielsweise die aktuelle *Epoch* verwendbar, wenn nur eine Challenge je Zeiteinheit erzeugt wird, z. B. bei einer Epoch-Auflösung von Nanosekunden. Allerdings ermöglicht die ausschließliche Verwendung eines Zeitstempels dem Angreifer die Vorbereitung von Responses für zukünftige Challenges. Außerdem könnte er bei Manipulation der Serverzeit vergangene Responses erneut einreichen. Deshalb sollten zeitbasierte Challenges mit Zufallszahlen kombiniert werden. Mit r als Zufallszahl und t als Zeitstempel sind Challenges als $t \parallel r$ konstruierbar. Wenn kein *CSPRNG* zur Verfügung steht, lassen sich Challenges unter Einbezug eines Server-Secret s als $hmac(t \| r, s)$ erzeugen. Die Kollisions-Wahrscheinlichkeit ergibt sich dann aus $|hmac|$.

Die in Kapitel 3.2 erarbeiteten Protokolle erlauben einen Pass-The-Hash-Angriff: Ein *MITM* kann unter Umgehung der Transportverschlüsselung das am Client gehashte Passwort abfangen und selbst zur Authentifizierung am Server einreichen. Durch Kopplung des Client-Hashings an einen dynamischen *CRAM* ist dies vermeidbar, da jede Response mit Passwort-Hashwert von der ausgestellten Challenge abhängt. Bedingung bleibt, dass der Verifier nur durch ihn ausgestellte Challenges ausschließlich einmalig akzeptiert.

3.3.2 Ausgewählte Protokolle mit bekannten Schwachstellen

Nachfolgend werden ausgewählte Challenge-Response-Protokolle mit mangelhaften Sicherheitsvorkehrungen vorgestellt. Durch Erläuterung der Eigenschaften und Schwachstellen der Protokolle sollen die Sicherheitsanforderungen an den zu konstruierenden *CRAM* herausgestellt werden. Dabei wird von den verwendeten Algorithmen und Datenformaten abstrahiert, da für die Betrachtung nur das Funktionsprinzip relevant ist. Gleiches gilt für den Einsatz zu kurzer und damit schwacher *Opaques* und Schlüssel.

CHAP

Das *Challenge Handshake Authentication Protocol (CHAP)* wird u. a. beim *Point-to-Point Protocol (PPP)* eingesetzt. Es löst das ältere *Password Authentication Protocol (PAP)* ab, welches Passwörter im Klartext überträgt. Wie in Abbildung 3.2 erkennbar, kann ein abgefangener Wert pw' nicht mehrfach verwendet werden, da er von *chal* abhängt. Der Speicherbedarf des Servers steigt, da für jede halboffene Verbindung die verwendete Challenge zu speichern ist. Wenn ein Angreifer *chal* kennt, kann er pw' angreifen. Die Sicherheit des Verfahrens hängt dann vom Kostenwert der Funktion *hash* ab. Ein Salting findet nur durch die Challenge statt. Ausschlusskriterium für das Protokoll ist, dass der Server *pw* im Klartext kennen muss.[Sim96]

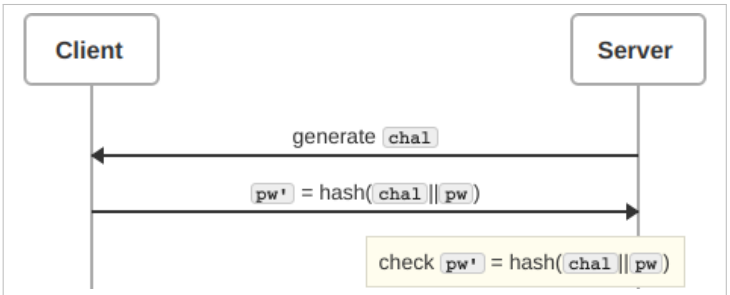


Abbildung 3.2: Ablauf des Protokolls CHAP
Quelle: Eigene Darstellung

NTLMv1

Das Protokoll *NT LAN Manager (NTLM)* Version 1 wurde von Microsoft entwickelt und diente dem Zugriff auf Netzwerkressourcen über die Windows-Anmeldedaten des Nutzers. Wie in Abbildung 3.3 dargestellt, wird der Passwort-Hashwert zur symmetrischen Verschlüsselung der Challenge genutzt. Dadurch muss der Server pw' anstatt *pw* kennen. Da pw' als Passwort-Äquivalent fungiert,

ist das Verfahren trotz *CRAM* anfällig für Pass-The-Hash-Angriffe. Zudem muss der Server pw' im Klartext speichern.

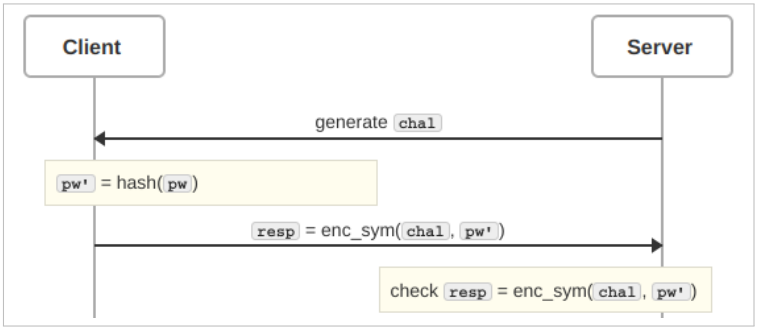


Abbildung 3.3: Ablauf des Protokolls NTLMv1
Quelle: Eigene Darstellung

MS-CHAPv2

MS-CHAPv2, spezifiziert in [Zor00], ist eine durch Microsoft entwickelte Erweiterung zum *CHAP*. Wie in Abbildung 3.4 erkennbar, wird eine gemeinsame Challenge von Server und Client gebildet. Dies bildet die Grundlage für eine wechselseitige Authentifizierung. Dabei prüft der Server regulär die Passwortangabe durch den Client. Zusätzlich produziert der Server eine Response pw'''' in Abhängigkeit von $chal_{cli}$ und pw'. Der Client durchläuft die gleichen Schritte zur Ermittlung von pw'''' und kann dadurch prüfen, ob auch der Server pw' kennt. Beide Parteien müssen sicherstellen, dass die gemeinsame Challenge mit dem durch die jeweilige Partei eingereichten Wert gebildet wurde, um Replay-Attacken zu verhindern. Bei diesem Verfahren bleiben die Probleme der Pass-The-Hash-Angriffe sowie Klartextspeicherung von pw' bestehen.

Digest Access Authentication

Dieses Protokoll kommt u. a. als Authentifizierungsverfahren für das *HTTP* zum Einsatz. Wie in Abbildung 3.5 dargestellt, fragt der *UA* eine geschützte Server-Ressource an. Zur Bindung an die Ressource werden angefragte *HTTP*-Methode und *Uniform Resource Identifier (URI)* des Requests in die Verarbeitung einbezogen. Der Server fordert den Client zur Authentifizierung auf. Das *opaque* dient der Zuordnung der Response zum Request. Zur Challenge-Bildung verwendet der Server *nonce* und der Client *cnonce*. Der Zähler *nc* ist bei Wahl einer neuen *cnonce* zu inkrementieren. Durch den Einsatz der Nonces kann eine Replay-Attacke verhindert werden. Dazu muss die Haltbarkeit einer Nonce zeitlich begrenzt sein oder sie ist nach Verwendung zu sperren. Der Server muss den aktuellen Wert *nc* vorhalten, um Mehrfachverwendungen zu erkennen.

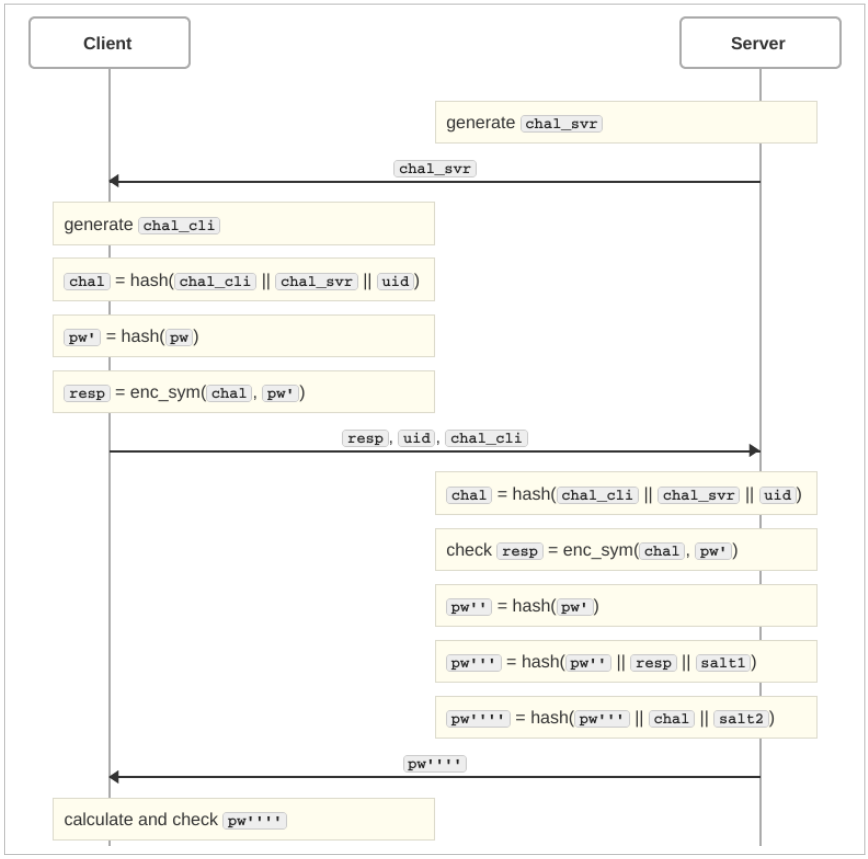


Abbildung 3.4: Ablauf des Protokolls MS-CHAPv2
Quelle: Eigene Darstellung

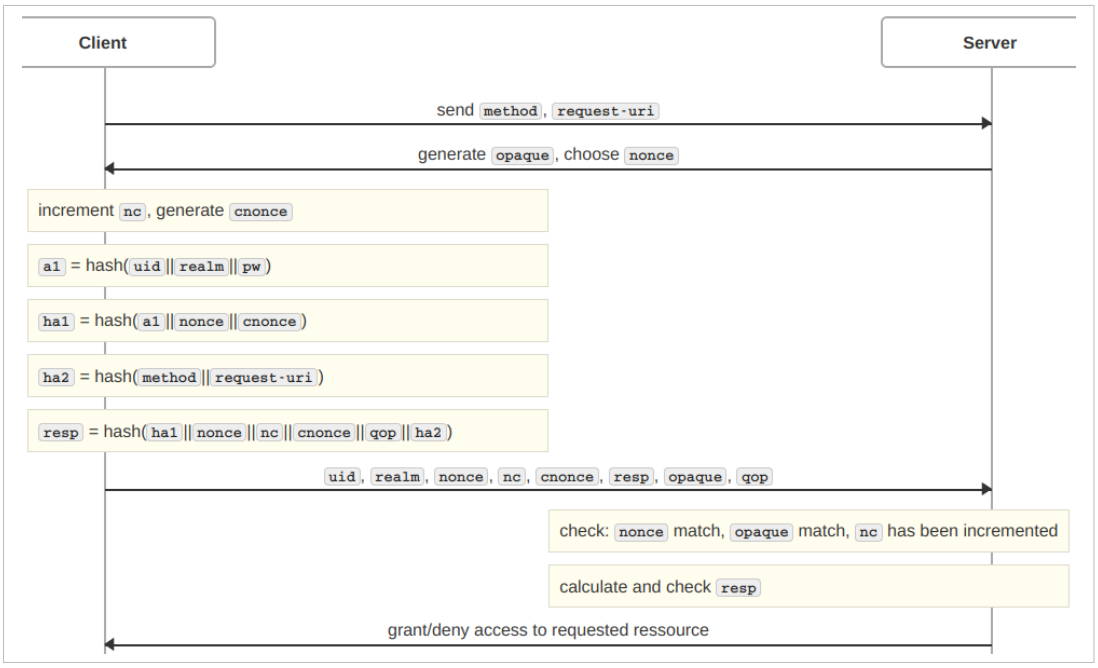


Abbildung 3.5: Ablauf des Protokolls zur Digest Access Authentication
Quelle: Eigene Darstellung

Der Wert *realm* wird einbezogen, um die Kombination von *uid* und *pw* verfeinert auf eine Fachlichkeit einzugrenzen, sodass z. B. nur für einen bestimmten Verwendungszweck Zugriff erteilt wird. Über den Wert *qop* kann eine zusätzliche Integritätssicherung angefordert werden, was irrelevant für die Betrachtung ist.

Eine optionale, wechselseitige Authentifizierung ist über die gemeinsame Challenge aus *nonce* und *cnonce* möglich. Dazu berechnet der Server im letzten Schritt:
$rspauth = hash(ha1 \parallel nonce \parallel nc \parallel cnonce \parallel qop \parallel hash(request\text{-}uri))$
Mit der Abschluss-Nachricht wird *rspauth* an den Client übermittelt, welcher dadurch prüfen kann, dass der Server $a1$ kennt.

Der Server kann $a1$ anstelle von *pw* vorhalten. Das Salting für $a1$ erfolgt mittels *realm* und *uid*. Beide Werte könnten einem Angreifer jedoch bekannt sein, sodass kein Peppering stattfindet. Weiterhin greift der Pass-The-Hash-Angriff und $a1$ fungiert als Secret, welches im Klartext auf dem Server vorliegt.

Ein *Eavesdropper* kann alle Werte des Systems abfangen oder erzeugen, die nicht von *pw* abhängen. Daraufhin kann er ein Dictionary-Angriff mit folgender, aufgelösten Formel und *pw* als einziger Unbekannten starten:
$resp = hash(hash(hash(uid \parallel realm \parallel pw) \parallel nonce \parallel cnonce) \parallel nonce \parallel nc \parallel cnonce \parallel qop \parallel hash(method \parallel request\text{-}uri))$
Deshalb ist der Einsatz einer langsamen Hashfunktion obligatorisch. Das Protokoll ist allerdings nur für schnelle Hashfunktionen definiert. Die Hashberechnungen stellen damit kein effektives Zeithindernis dar. Die Autoren der aktualisierten Spezifikation [SYAB15] geben als Mitigation eine empfohlene Mindestentropie für Passwörter von 128 Bit an. Dies lässt sich allerdings nur bei Verwendung als Passwortbesitz sinnvoll umsetzen. Ebenso empfehlen sie in [SYAB15] den dringenden Einsatz von *TLS*. Über diesen gesicherten Kanal wäre jedoch auch eine Klartext-Übertragung des Passwortes ähnlich sicher, wie sie durch *HTTP* Basic Auth nach [FHBS+99] geboten wird. Damit erhöht das Protokoll Softwarekomplexität und Speicherbedarf bei geringfügigem Sicherheitsgewinn. Dies dürfte ein Grund sein, warum das Protokoll selten verwendet und mitunter nur teilweise implementiert wird, siehe exemplarisch [Ngi19].
Ein weiteres Problem besteht im Kontext browserbasierter Applikationen: Weil Digest Auth durch *HTTP* implementiert wird, ist der *UA* für die Authentifizierung zuständig. Da der Web-Standard hierzu keine *API* für die Webseite bietet, sind grundlegende Funktionen wie Logouts nur über Hacks möglich.

3.3.3 Salted Challenge Response Authentication Mechanism

Dieses Verfahren baut auf dem in Kapitel 3.2 erarbeiteten Protokoll auf und ergänzt es zum *CRAM*. Funktionsweise, Sicherheitsaspekte und Integration in die *IDP*-Implementierung werden nachfolgend erläutert.

Entitäten und Ablauf des Protokolls

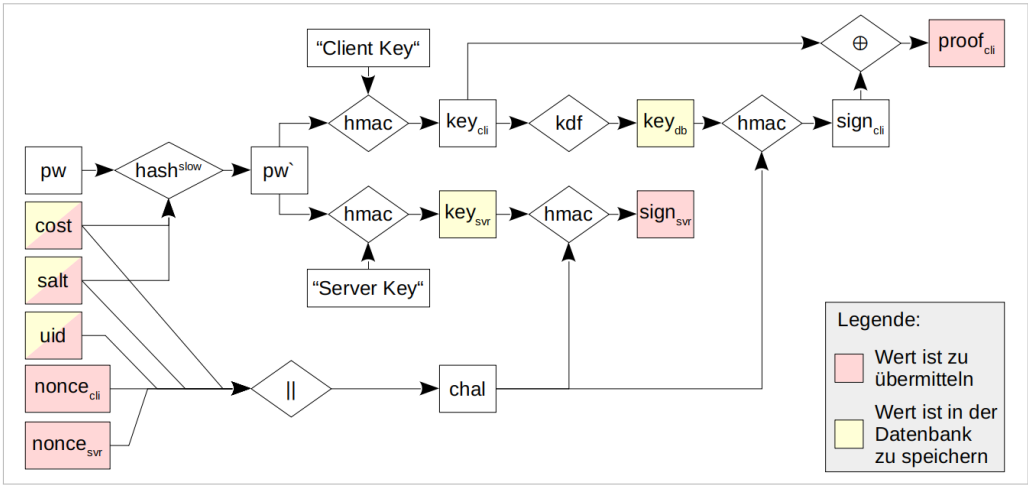


Abbildung 3.6: Zusammensetzung der Entitäten des SCRAM-Protokolls
Quelle: Eigene Darstellung

Das Verfahren *Salted Challenge Response Authentication Mechanism (SCRAM)* wurde in [MSMWN10] und [Han15] spezifiziert. Abbildung 3.6 zeigt die Entitäten des Systems.

Das aufwändige Hashing $hash^{slow}$ erfolgt durch den Client mit den Parametern *cost* und *salt*. Die Spezifikation sieht dazu einzig *PBKDF2* vor. Jedoch lässt sich die *SCRAM*-Konstruktion genauso mit einer zeit- und speicherintensiven Hashfunktion nutzen, was entsprechend Kapitel 2.2.3 präferiert wird.

HMAC wird als *KBKDF* eingesetzt, um aus pw' zwei Schlüssel abzuleiten. Dazu werden als *HMAC*-Salts zwei statische, ungleiche Strings genutzt, um unterschiedliche Schlüssel zu generieren.

Zur wechselseitigen Authentifizierung weisen Client und Server mittels key_{cli} sowie key_{svr} ihre Kenntnis zu pw gegenüber der jeweils anderen Partei nach. Dazu dienen die Signaturen $sign_{cli}$ und $sign_{svr}$. Diese werden mittels *HMAC* abhängig von der Challenge erzeugt und sind damit für jeden Authentifizierungsvorgang individuell. Client und Server bestimmen die Challenge *chal* gemeinsam aus den jeweils gewählten Werten $nonce_{cli}$ und $nonce_{svr}$. Zusätzlich wird die Challenge an *uid*, *cost* und *salt* durch Aufnahme in die Konkatenation gebunden.

Die Client-Signatur $sign_{cli}$ wird nicht direkt aus key_{cli} abgeleitet, sondern aus dem in der *DB* gespeicherten Hashwert key_{db}. Der Übergang von key_{cli} zu key_{db}

entspricht funktional dem Server-Hashing aus Kapitel 3.2.1. Ein Angreifer mit Kenntnis von key_{db} könnte $sign_{cli}$ bestimmen und damit das Opfer impersonieren. Deshalb wird anstatt der Signatur selbst ein Wert $proof_{cli}$ zur Übertragung eingesetzt. Dieser ergibt sich aus einer XOR-Verknüpfung von key_{cli} und $sign_{cli}$, weshalb die Länge beider Werte bitgleich sein muss. Dieser Aufbau verhindert einen Pass-The-Hash-Angriff mittels key_{db}, ohne dabei key_{cli} zu exponieren.

Abbildung 3.7 stellt den Ablauf einer Anmeldung mittels *SCRAM* dar. Nach Empfang von $proof_{cli}$ prüft der Server die Client-Signatur, wozu er zunächst $sign_{cli}$ berechnet. Aufgrund der Invertierbarkeit des XOR-Operators kann der Server aus $sign_{cli}$ und dem vorgelegten $proof_{cli}$ den clientside-erzeugten key_{cli} bestimmen. Wenn die Ableitung von key_{cli} den hinterlegten key_{db} ergibt, ist der Client authentifiziert. Um sich zu authentifizieren, sendet der Server $sign_{svr}$ abschließend an den Client. Durch Berechnung von $sign_{svr}$ und Vergleich mit dem vorgelegten Wert kann der Client auch den Server authentifizieren.

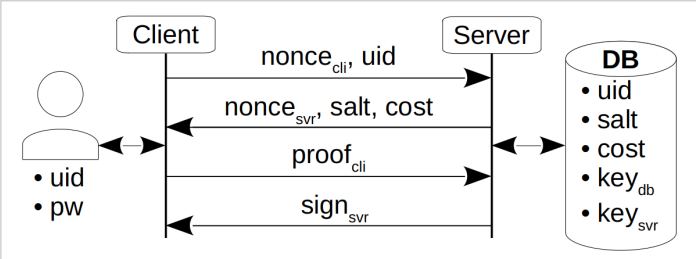


Abbildung 3.7: Ablauf einer Anmeldung mittels SCRAM-Protokoll
Quelle: Eigene Darstellung

Sicherheitsbetrachtung und Einbindung in die *IDP*-Anwendung

SCRAM ist nicht vom Pass-The-Hash-Angriff betroffen: Aus key_{db} müsste zuerst der hochentropische key_{cli} bestimmt werden. Wie in der weiteren Sicherheitsbetrachtung ausgeführt, ist dies durch das Hashing enorm aufwändig. Ein Angreifer kann einen abgefangenen Wert $proof_{cli}$ nicht wiederverwenden, da der Server neue, möglichst einmalige Werte für $nonce_{svr}$ wählt.

SCRAM erzielt ein *Server-Relief*, da nur der Client $hash^{slow}$ berechnen muss. Die durch den Server auszuführenden Hashfunktionen sind performant.

SCRAM ermöglicht eine wechselseitige Authentifizierung. Allerdings wird key_{svr} im Klartext persistiert. Ein Angreifer könnte nach Erlangung des Wertes den Server impersonieren. Dazu muss er Nutzer motivieren sich am falschen Server anzumelden. Aus der Wahrscheinlichkeit hierzu ergibt sich das Risiko der Schwachstelle. Der Schutz von key_{svr} wird im Weiteren diskutiert.

SCRAM sieht kein Server-Salting zur Berechnung von key_{db} vor, was aber ergänzbar wäre. Allerdings müsste dieses Salt ebenfalls dem Client mitgeteilt werden, da dieser auch key_{db} berechnen muss. Jedoch fließt ein Client-Salting bereits in das langsame Hashing ein. Deshalb ist ein Server-Salting nicht vorteilhaft und ein Server-Peppering nicht möglich. Lediglich ein Client-Peppering wäre zu den in Kapitel 3.2.4 erläuterten Bedingungen ergänzbar.
Anstelle eines Server-Peppering wird eine symmetrische Verschlüsselung von key_{db} und key_{svr} mit einer Funktion enc_{sym} sowie einem Secret key_{pep} implementiert. In der *DB* werden dann $enc_{sym}(key_{db}, key_{pep})$ und $enc_{sym}(key_{svr}, key_{pep})$ abgelegt. Wie beim Peppering ist key_{pep} außerhalb der *DB* aufzubewahren und geheim zu halten. Beim Einsatz einer Blockchiffre sollte die Länge der Eingabedaten (key_{db} und key_{svr}) ein Vielfaches der Blockgröße von enc_{sym} betragen. Andernfalls müsste ein Padding angewendet werden, was ein Angreifer mit *DB*-Einblick zur Einschränkung des Suchraums nutzen könnte.

SCRAM verhindert eine Einsicht des Passwortes durch den Server. Zur Registrierung werden nur key_{svr} und key_{db} übermittelt. Da key_{cli} nicht zu senden ist, besteht auch hier für einen *Eavesdropper* keine Möglichkeit zum Pass-The-Hash. Allerdings ließe sich key_{svr} dann für den skizzierten Angriff mit falschem Server missbrauchen. Dagegen ist folgendes Vorgehen anwendbar: Der Client übermittelt zur Registrierung $(key_{db}, enc_{sym}(key_{svr}, key_{cli}))$. Zur Anmeldung erfährt der Server key_{cli} und kann damit die Entschlüsselung vornehmen.
Wie jedoch in [MSMWN10, S. 20 ff.] angemerkt, ersetzt auch *SCRAM* nicht den obligatorischen Einsatz von *TLS*. Der Schutz durch wechselseitige Authentifizierung bei Server-Kompromittierung ist Diskussionsgegenstand von Kapitel 5.1.3.

Verfügt ein Angreifer über abgefangene oder gespeicherte Werte entsprechend Abbildung 3.6, so sind weiterhin Angriffe gegen die Hashing-Verfahren möglich:

1. Angriff auf key_{db}, key_{svr}, $sign_{svr}$ oder $proof_{cli}$ zur Bestimmung von pw
2a. Angriff auf key_{db} oder $proof_{cli}$ zur Bestimmung von key_{cli}
2b. Angriff auf key_{svr} oder $sign_{svr}$ zur Bestimmung von pw'

Die Punkte 2a und 2b stellen Variationen des gleichen Angriffs gegen hochentropisches Schlüsselmaterial dar. Die Analyse aus Kapitel 3.2.1 bezüglich der dort vorgestellten Angriffe 1 und 2 ist damit direkt auf die hier aufgelisteten Angriffe übertragbar. Daraus folgt, dass für die *KBKDFs* eine hohe Ausgabelänge zu wählen ist, z. B. 256 Bits. Für das Client-Salting gilt die Betrachtung aus Kapitel 3.2.2 sowie 3.2.4. Bei korrekter Parameterwahl hängt die Sicherheit des Systems gegen vorberechnete Hashwerte damit einzig von der Passwort-Stärke und dem Ressourcenaufwand der $hash^{slow}$ ab.

Die *IDP*-Realisierung folgt der Betrachtung aus Kapitel 3.2.2: Das Account-Salt wird aus einem Applikations-Salt gebildet. Ebenso ist der Wert *cost* für alle Accounts gleich. Eine Rotation der Parameter kann wie vorgestellt ergänzt werden. Damit entfällt eine Speicherung von *salt* und *cost* je Account in der *DB*.

Im letzten Protokollschritt sendet der Server $sign_{svr}$ an den Client, siehe Abbildung 3.7. Zu diesem Zeitpunkt ist der Client bereits authentifiziert. Deshalb kann der Server weitere Response-Parameter beifügen. Dadurch lässt sich mitunter ein zusätzlicher Request einsparen. Wird *SCRAM* zur Anmeldung ausgeführt, so können beispielsweise die Session-Tokens angehängt werden, siehe Kapitel 2.3.3.

In [Mel16, S. 10 ff.] wird eine *SCRAM*-Reauthentifizierung innerhalb eines *Roundtrips* vorgeschlagen. Eine einmal etablierte Challenge wird fixiert und um einen jeweils zu inkrementierenden Nonce-Counter erweitert. Der Client hält key_{cli} und key_{svr} im Cache vor und muss deshalb nur einmalig $hash^{slow}$ ausführen.
Von diesem Verfahren wird abgeraten: Es ist nur vorteilhaft, wenn der Client das Passwort-Äquivalent key_{cli} im Klartext vorhält. Stattdessen sollten Bearer-Tokens mit geringeren Berechtigungsstufen eingesetzt werden, siehe Kapitel 2.2.4.

SCRAM-PLUS

Das bisher vorgestellte *SCRAM*-Protokoll besitzt eine Designschwäche: Der Server muss zwischen erstem und zweitem Request die Challenge zwischenspeichern, siehe Abbildung 3.7. Andernfalls könnte er nicht mehr überprüfen, ob die Challenge aus $nonce_{svr}$ erzeugt wurde. Ein Angreifer wäre in der Lage einen eigenen Wert zu verwenden. Durch diese „halb offenen“ Verbindungen entsteht im Backend unauthentifizierter, temporärer Zustand, den es zu vermeiden gilt.
Der Server könnte den Zustand an den Client auslagern. Dazu stellt er einen clientgestützten Bearer-Token aus, siehe Kapitel 2.2.4. Dieser enthält die *SCRAM*-Challenge und ist durch seine Signatur für den Server überprüfbar. Allerdings hat der Server damit keine Möglichkeit zwischen gültigen *SCRAM*-Tokens eines Accounts zu unterscheiden. Eine Replay-Attacke für einen abgefangenen Wert $proof_{cli}$ ist möglich. Zur Wahrung der Nonce-Eigenschaft muss der Server zwischen den Requests zumindest $nonce_{svr}$ vorhalten und nach Einsendung des zugehörigen Bearer-Tokens oder Ablauf der Token-Lifetime löschen.

Der bisherige Ansatz bindet den Zustand an Client sowie Server. Damit besteht ein Restrisiko, dass der Zustand einer Verbindung in einer anderen genutzt wird. Dazu könnte ein Angreifer z. B. einen Implementierungsfehler zur Replay-Attacke ausnutzen, um verschlüsselte Datenpakete aufzuzeichnen und in einer eigenen Verbindung zum Server wieder einzuspielen.

Optimalerweise müsste gegen derartige Angriffe der Zustand an die Verbindung zwischen Client und Server gebunden werden. *HTTP* als zustandsloses Protokoll kann dies nicht leisten. Deshalb nutzen die in [MSMWN10] als SCRAM-PLUS spezifizierten Verfahren eine andere Technik: Der *SCRAM*-Übertragung liegt *TLS* zugrunde. *TLS* ist im Gegensatz zu *HTTP* nicht zustandslos. Tatsächlich kann *TLS* nicht zustandslos sein, da eine Verbindung zumindest auf dem ausgehandelten *Diffie-Hellman-Key* aufbaut. Jede Verbindung besitzt damit einen einzigartigen Fingerprint. Dieser wird im Sinne des Schichtenmodells normalerweise nicht an andere Netzwerk-Layer propagiert. Jedoch erlaubt die Aufnahme des *TLS*-Fingerprints in die Bildung der *SCRAM*-Challenges eine Kopplung an die Client-Server-Connection. Damit ist eine *SCRAM*-Challenge in anderen Connections für einen Angreifer nicht mehr verwendbar. Diese Bindung wird in [Wil07] als „unique channel binding“ aufgeführt und in [AZW10] konkreter als „tls-unique“ definiert.
Eine Alternative stellt „tls-server-end-point“ nach [AZW10] dar: Hierbei wird nur das Server-Zertifikat zur Fingerprint-Bildung genutzt. Dadurch wird gegenüber dem Client zumindest die Kommunikation mit dem gleichen Backend nachgewiesen. Die erzielte Schutzwirkung lässt sich daher mit dem Einsatz von *HPKP* vergleichen, siehe Kapitel 2.2.1. Da jedoch nicht jeder *UA* *TLS*-Channel-Binding unterstützt, muss eine Abwärtskompatibilität hergestellt werden. Zur Vorbeugung sich daraus ergebender Downgrade-Attacken gilt es Verbindungen auf Applikationsebene bezüglich ihre Unterstützung von Channel-Binding festzulegen.

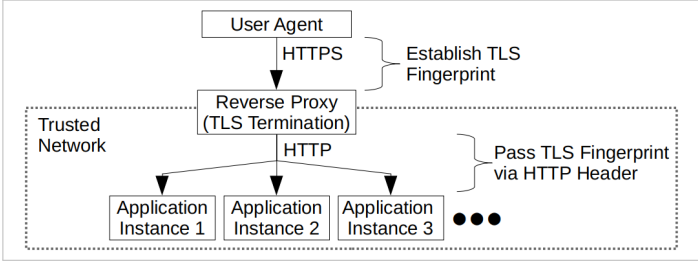


Abbildung 3.8: Mögliche Einbindung von SCRAM mit TLS Channel Binding in die Backend-Architektur mit *TLS*-Terminierung am Reverse-Proxy
Quelle: Eigene Darstellung

Ein weiteres Problem besteht in der architektonischen Backend-Realisierung. Wie in Kapitel 2.1.1 erläutert, nimmt der Reverse-Proxy die *TLS*-Terminierung vor. Applikations-Server können *TLS*-Fingerprints nur erfahren, wenn der Reverse-Proxy sie weiterleitet. Dies kann, wie in Abbildung 3.8 skizziert, über einen *HTTP*-Header geschehen. Setzt ein Client als Angreifer diesen Header, so muss er vom Reverse-Proxy überschrieben werden, bevor der Request weitergeleitet wird. Damit das Verfahren so implementierbar ist, muss der Reverse-Proxy das Chan-

nel-Binding unterstützen und umsetzen. Da dies nicht zu den Standard-Aufgaben eines Reverse-Proxies zählt, ist die Umsetzbarkeit von Software und Organisation abhängig. Bei skizzierter Backend-Architektur bietet *SCRAM* den Vorteil, dass die Vertrauenskette nicht am Reverse-Proxy mit der *TLS*-Terminierung endet, sondern bis in die Instanzen der Applikationsserver ausgedehnt wird, siehe Abbildung 3.8. In Kapitel 4.3.1 wird mit dem Token Binding eine Alternative zum Channel Binding vorgestellt.

3.3.4 Risiko-Bewertung

Nachfolgend wird eine Risiko-Bewertung zum Client-Hashing mit *CRAM* vorgenommen. Die Risiko-Bewertung basiert auf der Angreifer-Klassifikation aus Kapitel 2.1.2 und nutzt das Threat-Modelling-Verfahren DREAD-D, welches in Kapitel 2.1.3 erläutert wurde.

Für die schwachen *CRAM*-Protokolle aus Kapitel 3.3.2 wurden folgende Risiken bereits bewertet:

- Möglichkeit zum Pass-The-Hash-Angriff trotz geschützter *TLS*-Verbindung: Mittleres Risiko nach Tabelle 3.3 in Kapitel 3.1.3.
- Klartext-Speicherung von Passwort-Äquivalenten im Backend: Hohes Risiko nach Tabelle 3.5 in Kapitel 3.1.3.

Der Einsatz von *CRAM* bedarf des temporären Haltens von Challenges als unauthentifizierter Zustand im Backend. Hieraus ergibt sich eine *DDOS*-Möglichkeit, in deren Folge eine Server-Überlastung und damit potenziell ein Server-Ausfall möglich ist, siehe Tabelle 3.23.
Auch ohne *CRAM* muss der Server durch den Einsatz von *TLS* für jede Verbindung einen Zustand halten. Die Risiko-Bewertung ist daher ebenso auf *DDOS*-Angriffe gegen *TLS*-Verbindungen anzuwenden. Da die mögliche Zahl parallel belegbarer Sockets je Host begrenzt ist, können legitime Nutzer während des Angriffs im Zweifelsfall keine Verbindung zum Server aufbauen, siehe Tabelle 3.22. Wie aus dem Vergleich von Tabelle 3.22 und 3.23 hervorgeht, steigern *CRAM*-Protokolle das ohnehin vorhandene Risiko nur geringfügig.

Eine wechselseitige Authentifizierung findet auch ohne den Einsatz von *CRAM* bereits statt: Der Server muss ein vertrauenswürdiges *TLS*-Zertifikat vorweisen und der Client den Passwort-Hash. Das Risiko einer Fälschung des Server-Zertifikats ist von den *Technischen und Organisatorischen Maßnahmen (TOM)* der *PKI* abhängig, siehe Kapitel 2.2.1. Sie sollten im Interesse des Dienstbetreibers ein äußerst hohes Niveau besitzen, siehe Tabelle 3.24. Währenddessen ist ein

Tabelle 3.22: Risiko der Server-Überlastung mittels DDOS-Attacken durch TLS-Einsatz

D	1/5	Eingeschränkte Verfügbarkeit solange der Angriff anhält
R	4/5	*TLS*-Verbindungen aufbauen und lange offen halten
E	2/5	Umgehung Rate-Limitings mit hoher Request-Anzahl
A	4/5	Betrifft sämtliche im Zeitraum aktiven Nutzer
$\sum$	**11/20**	⇒ **Mittleres Risiko**

Tabelle 3.23: Risiko für CRAM zur Server-Überlastung mittels DDOS-Attacken

D	1/5	Eingeschränkte Verfügbarkeit solange der Angriff anhält
R	5/5	Anfragen zur Ausgabe von Challenges senden
E	2/5	Umgehung Rate-Limitings mit hoher Request-Anzahl
A	4/5	Betrifft sämtliche im Zeitraum aktiven Nutzer
$\sum$	**12/20**	⇒ **Mittleres Risiko**

Erraten oder Bekanntwerden von Passwörtern einzelner Nutzer alltäglicher und wahrscheinlicher, siehe Tabelle 3.25. Gegenüber einer einfachen Hash-Übertragung vom Client zum Server sichert die wechselseitige Authentifizierung durch *CRAM* die Server-Echtheit zusätzlich ab. Diese sollte aber durch die getroffenen *TOM* bereits hoch sein, sodass der Sicherheitsgewinn gering ist. Die Nutzer-Authentifizierung profitiert hiervon nicht, was allerdings den begrenzten Sicherheitseigenschaften des Merkmals Passwort geschuldet ist. Dieser Mangel wird in Kapitel 4 durch Einführung einer *MFA* beseitigt.

Tabelle 3.24: Risiko der Kompromittierung des Server-Zertifikats

D	5/5	Kompromittierung der Server-Kommunikation
R	1/5	Abhängig von ergriffenen TOM
E	1/5	Abhängig von ergriffenen TOM
A	5/5	Betrifft alle bis zur Feststellung aktiven Nutzer
$\sum$	**12/20**	⇒ **Mittleres Risiko**

Tabelle 3.25: Risiko der Kompromittierung eines Accounts

D	5/5	Kompromittierung des betroffenen Accounts
R	4/5	Passwort ist erratbar oder entwendbar, z. B. via Phishing
E	4/5	Angriff gegen Nutzer, z. B. Phishing-Kampagne
A	2/5	Angriff kann gegen mehrere Nutzer gerichtet sein
$\sum$	**15/20**	⇒ **Hohes Risiko**

SCRAM macht ein Server-Salting obsolet. Da das Server-Hashing nur aus hochentropischen Eingabe-Hashes erfolgt, ergibt sich hieraus kein Threat, siehe Kapitel 3.3.3. Das Risiko zum Angriff mit vorberechneten Hashwerten ist stattdessen anhand der Verschlüsselung als Peppering-Ersatz zu beurteilen, siehe Tabelle 3.26 und 3.27. Hieraus folgt, dass *SCRAM* im Falle eines Data-Breaches von *DB* und Peppern keinen starken Schutz vor Angreifern bietet, die gezielt einen hohen Aufwand zur Vorbereitung von Hashwerten betrieben haben.

Tabelle 3.26: Risiko für SCRAM durch Angreifer mit DB-Einblick ohne Peppers

D	1/5	Kenntnis der verschlüsselten Passwort-Hashwerte
R	1/5	Aufwändiges Hashing nötig, ohne Ergebnis-Überprüfung
E	2/5	Zugang zu DB-Auszug notwendig
A	3/5	Betrifft einen Account, aber Angriff parallelisierbar
$\sum$	**7/20**	⇒ **Niedriges Risiko**

Tabelle 3.27: Risiko für SCRAM durch Angreifer mit DB-Einblick und Peppers

D	5/5	Übernahme von Accounts der Anwendung
R	2/5	Aufwändiges Hashing in Vorbereitung nötig
E	1/5	Zugang zur Server-Infrastruktur
A	3/5	Betrifft einen Account, aber Angriff parallelisierbar
$\sum$	**11/20**	⇒ **Mittleres Risiko**

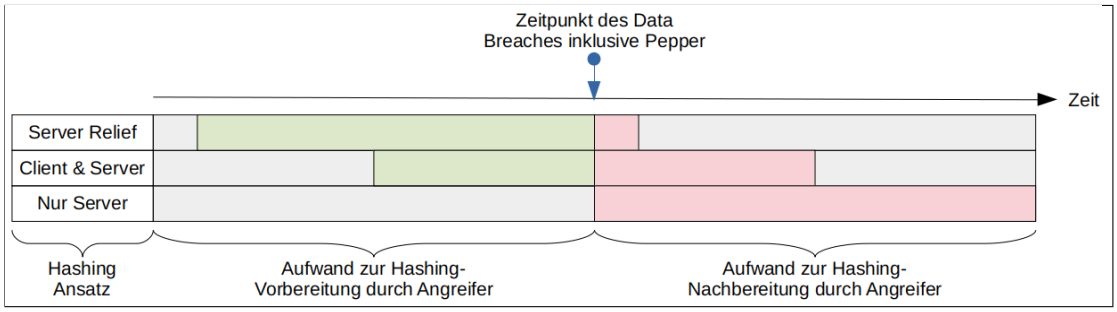


Abbildung 3.9: Zeitliche Verteilung von Vorbereitung und Nachbereitung eines Data-Breaches bezüglich Hashing durch einen Angreifer
Quelle: Eigene Darstellung

Zur Beurteilung dieses Angriffs muss der Hashing-Aufwand der Protokoll-Varianten verglichen werden. Abbildung 3.9 fasst die Betrachtung aus Kapitel 3.2 bezüglich der Zeitverteilung zusammen. Wenn der Angreifer im Besitz der *DB*-Inhalte sowie der Pepper ist, so kann er durch Hashing-Versuche das richtige Passwort zu einem Account bestimmen. Für den Client-Hashaufwand kann dies in Vorbereitung auf einen *Data Breach* erfolgen und für das Server-Hashing in Reaktion darauf.

Wenn der Angreifer keine Hashwerte vorbereitet hat, so sind alle drei dargestellten Varianten gleich stark bezüglich der Nachbereitung. Wenn der Angreifer Hashwerte vorbereitet hat, so ist die notwendige Zeitspanne zur Nachbereitung für Server-Relief-Hashing besonders kurz. Hieraus in der Risiko-Bewertung direkt eine erhöhte Gefahr abzuleiten, würde zu kurz greifen: Ein Data-Breach muss nicht immer direkt bemerkt werden. Mitunter hat ein Angreifer viel Zeit zur Nachbereitung, genauso wie er viel Zeit zur Vorbereitung haben kann.

Wird ein Angriff direkt bemerkt, so ist eine Nachbereitung nur in der Zeitspanne zwischen dem Zeitpunkt des Data-Breaches und der Einleitung von Maßnahmen durch den Dienstbetreiber möglich. Eine Maßnahme sollte die Sperrung der Passwort-Authentifizierung und Ersetzung durch einen einmaligen Login via Zu-

rücksetzung der Anmeldedaten sein. In diesem Szenario wäre ein Server-Relief-Hashing tatsächlich gefährdeter als ein ausschließliches Server-Hashing.
Allerdings ist die Betrachtung unvollständig, da der Angreifer sowohl *DB*-Inhalt als auch die Pepper kennen muss. In einem einfachen Data-Breach sollte das nicht möglich sein, da die Pepper getrennt verwahrt werden, siehe Kapitel 2.2.3. Ein Angreifer muss damit schon umfassenden Zugang zur Server-Infrastruktur besitzen, um den Angriff ausüben zu können. In dieser Position könnte er aber vermutlich auch schon die Applikation selbst manipulieren.
Kennt der Angreifer jedoch nicht die Pepper, so ist weiterhin ein starker Schutz bei Server-Relief-Hashing und Server-Hashing gegeben, vergleiche Tabelle 3.26 und Tabelle 3.1 in Kapitel 3.1.3.

Beurteilung *SCRAM* verbessert die Sicherheit clientseitiger Passwortverarbeitung mit *Server-Relief* auf einem sehr hohen Niveau. Das *DDOS*-Problem ist kalkulierbar und über Rate-Limiting einzudämmen, siehe Kapitel 5.1.5.
Wenn der Schutz durch *TLS* als ausreichend angenommen wird, so ist der Overhead durch *SCRAM* vermeidbar. Stattdessen kann ein einfaches Client-Hashing betrieben werden. Probleme wie Pass-The-Hash- und Replay-Attacken deckt dann der *TLS*-Einsatz ab. Da *SCRAM* abgesehen von der inhärenten *DDOS*-Möglichkeit jedoch keine Sicherheitsprobleme einführt, ist eine Implementierung von Server-Relief-Hashing über die *SCRAM*-Konstruktion zu empfehlen.

Zusammenfassend ist eine sichere, clientseitige Passwortverarbeitung mit den vorgestellten Techniken realisierbar. Einzig der Angriff mit vorberechneten Hashwerten lässt sich ohne Client-Peppering nicht verhindern. Falls es für einen Angreifer lukrativ sein könnte, diesen Aufwand zur Vorbereitung von Hashwerten für einen Account zu betreiben, so reicht eine clientseitige Passwortverarbeitung als Schutzmaßnahme für die Authentifizierung nicht aus. Allerdings lohnt sich ein derart hoher Aufwand für einen Angreifer nur, wenn die Übernahme des Accounts für ihn äußerst wertvoll ist. In dieser Konstellation ist der alleinige Einsatz eines Passwortes zum Schutz eines so wichtigen Accounts kritisch zu hinterfragen. Dann sollte der Systementwurf bereits eine ergänzende *MFA* vorsehen, welche das nächste Kapitel behandelt.

4 Mehrfaktor-Authentifizierung

Zur *MFA* müssen Nutzer im Gegensatz zur *1FA* verschiedene Merkmale zur Authentifizierung vorweisen. Eine bekannte Form ist die *Zweifaktor-Authentifizierung (2FA)*. Nachfolgend wird die Einbindung einer *MFA* in das bisher erarbeitete *IDP*-Protokoll vorgestellt. Anschließend werden symmetrische und asymmetrische *MFA*-Verfahren diskutiert. Bei symmetrischen Verfahren teilen Client und Server ein Geheimnis in unterschiedlichen Verarbeitungsstufen, z. B. Klartext oder Hashwert. Hierzu zählt die Authentifizierung mittels Passwort. Im Gegensatz dazu nutzen asymmetrische Verfahren als Primitiv entsprechend asymmetrische Kryptographie, um Schlüsselmaterial zwischen Client und Server aufzuteilen.
Ausgehend von Kapitel 3 wird als erster Faktor eine passwortbasierte Authentifizierung vorgegeben. Die weiteren Faktoren stammen aus der Kategorie Besitz. Biometrische Faktoren werden in Kapitel 4.3.3 einbezogen.

4.1 Einbindung in das Protokoll

Nachfolgend werden Konzepte und Methoden erläutert, das Referenz-Protokoll aus Kapitel 2.3 um eine optionale *MFA* anzureichern.

4.1.1 Client-Peppering via Zweifaktor-Authentifizierung

Angenommen, eine bestehende *1FA*-Implementierung soll um eine *MFA* erweitert werden, wobei nur der Client-Code anzupassen ist. Eine Realisierung könnte zwei verschiedene Passwortfelder für Passwortkenntnis und -besitz vorsehen. Der Nutzer müsste das erste Passwort pw_1 eingeben (Kenntnis) und das zweite pw_2 aus einer anderen Quelle, wie einem Passwort-Manager, einfügen (Besitz). Daraufhin würde der *UA* sie konkatenieren und $pw_1 \| pw_2$ an den Server senden. Aus Client-Perspektive wäre dies eine *2FA*, aus Server-Sicht eine *1FA*. Effektiv fungiert der Passwortbesitz als Pepper für die Passwortkenntnis.
Ausgehend von einem Alphabet Σ seien $pw_1 \in \Sigma^*$ und $pw_2 \in \Sigma^*$ mit $|pw_1| > 0$ und $|pw_2| > 0$. Der resultierende Passwortwert $pw_1 \| pw_2$ wäre deutlich sicherer als die ausschließliche Verwendung von Passwortkenntnis, da $|\Sigma^{|pw_1|}| < |\Sigma^{|pw_1|+|pw_2|}|$.
Der Schutz gegen Offline-Angriffe auf den Passwort-Hash in der *DB* ist hoch. Allerdings profitiert der Online-Schutz hiervon nicht: Ein Angreifer kann unter Umgehung des Rate-Limitings unbegrenzt viele Versuche durchführen. Eine Account-Sperrung nach zu vielen Fehlversuchen ist nicht umsetzbar. Andernfalls könnte ein Angreifer durch Falscheingaben beliebige Accounts sperren.

Deshalb wird stattdessen ein zweistufiger Prozess präferiert, bei dem die Faktoren nacheinander vorgelegt werden. Der Offline-Schutz fällt geringer aus, da $|\Sigma^{|pw_1|}| + |\Sigma^{|pw_2|}| \leq |\Sigma^{|pw_1|+|pw_2|}|$. Jedoch wird der Online-Schutz verbessert: Ein Angreifer muss, um den zweiten Faktor zu erraten, auch bei jeder Anfrage den korrekten ersten Faktor am Server vorlegen. Durch diese Authentifizierung kann der Server den Account nach zu vielen Fehlversuchen sperren. Dieser Ansatz wird im nächsten Abschnitt zur iterativen *MFA* ausgebaut.
Die Architektur bedingt damit eine Ausweitung und Unterstützung des *MFA*-Prozesses durch das Backend. Im Gegensatz dazu kann ein *MFA*-Peppering, wie oben vorgestellt, rein clientseitig erfolgen. Reine Client-*MFA*-Verfahren ohne serverseitige Anteilnahme durch den *IDP*-Betreiber werden in Kapitel 4.2.1 vertieft.

4.1.2 Iterative Mehrfaktor-Authentifizierung

Für jeden Account wird eine Anzahl an Faktoren festgelegt, die zur Authentifizierung nachzuweisen sind. Diese Anzahl wird im Weiteren als *MFA*-Level bezeichnet. Ein Authentifizierungsvorgang, nachfolgend *MFA*-Versuch genannt, wird als mehrstufiger Prozess umgesetzt: Nach Anmeldung mit dem ersten Faktor ist der Nutzer authentifiziert, kann aber nicht auf die Anwendung zugreifen. Möchte er den Login abschließen, so muss er eine Authentifizierungsschleife durchlaufen. In jeder Iteration dieses *MFA*-Versuches legt der Client ein Merkmal am Server vor. Der Server führt zum *MFA*-Versuch einen Fehlerzähler, der bei falscher Antwort erhöht wird. Überschreitet der Zähler einen Schwellwert, ist der Prozess gescheitert. Hat der Client hinreichend viele Merkmale zur Erlangung des geforderten *MFA*-Levels nachgewiesen, wird die angeforderte Aktion gestattet. Eine optionale *MFA* (*MFA*-Level 1) wird ermöglicht, indem nach erfolgreicher *1FA* die Schleife übersprungen wird.
Der Schwellwert für den Fehlerzähler eines *MFA*-Versuches sollte leicht höher gewählt werden als das *MFA*-Level, da durch Menschen erbrachte Faktor-Nachweise einem Eingabefehler unterliegen können. Aus *UX*-Gründen sollte daher für diese eine begrenzt wiederholte Eingabe toleriert werden.

Die Anzahl an möglichen *MFA*-Versuchen für einen Account ist zu begrenzen, um Schutz gegen Angreifer mit Zugang zu einem Faktor zu bieten. Dabei zählt jeder angefangene Authentifizierungsversuch, da ein *MFA*-Versuch nicht beendet werden muss. Ein Account wird als gesperrt markiert, wenn zu viele fehlgeschlagene oder unvollendete *MFA*-Versuche vorliegen. Bei erstmaligem Überschreiten dieses Limits sollte der Nutzer per Mail gewarnt werden. Abbildung 4.1 fasst das Vorgehen zusammen.

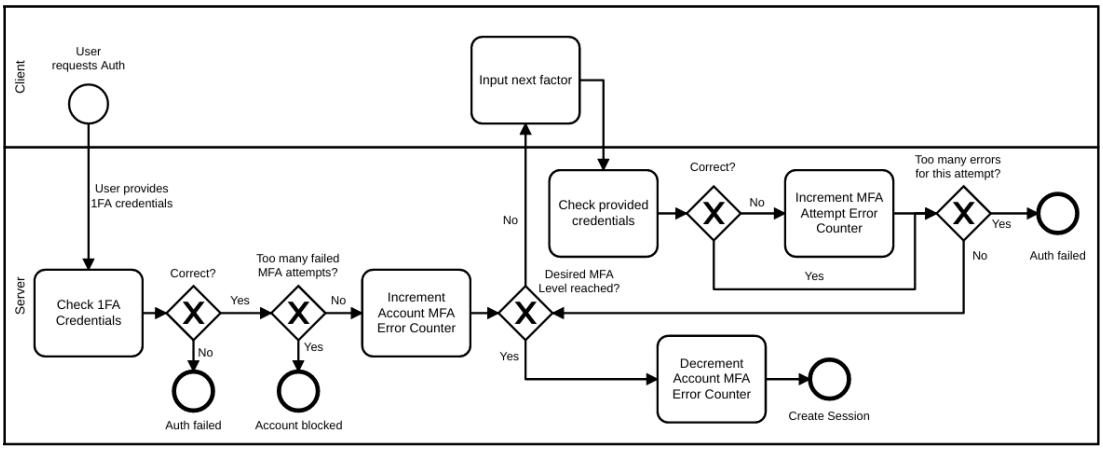


Abbildung 4.1: Protokoll zur optionalen, iterativen *MFA* mit begrenzten Versuchen
Quelle: Eigene Darstellung

Da ein Faktor nach erfolgreicher Verwendung nicht nochmals vorzeigbar sein darf, muss zu jedem *MFA*-Versuch eine Liste bereits genutzter Faktoren vorgehalten werden. Zur Unterscheidung wird jedem Faktor durch den Server eine ID zugewiesen. Nach Absolvierung der *1FA* besteht bereits eine Art Sitzung zwischen Client und Server, deren Ziel der Abschluss der *MFA* ist. Das Konzept clientgestützter Bearer-Tokens aus Kapitel 2.2.4 ist damit einsetzbar. Nach erfolgreicher *1FA* sowie jedem Vorzeigen eines Faktors wird ein neuer *MFA*-Token ausgestellt. Dieser ist zusammen mit dem nächsten Faktor vorzuzeigen. Ein *MFA*-Token nimmt folgende Informationen auf:

- Nutzerkennung: Bindung an den Account, zu dem die *MFA* erfolgen soll
- Gültigkeitszeitraum: Steuerung der eingeräumten Dauer zur Erbringung des nächsten Nachweises eines bisher ungenutzten Faktors
- Fehlerzähler: Anzahl bisheriger Iterationen ohne gültigen Faktor-Nachweis
- Liste zuvor genutzter Faktoren: Verhinderung einer Mehrfachverwendung
- Zu autorisierende Aktion: z. B. Anmeldung, Initiierung einer Transaktion
- Nonce-Wert zum aktuellen *MFA*-Versuch (wird nachfolgend erläutert)

Nach seinem Einsatz ist ein *MFA*-Token zu invalidieren. Eine Speicherung bis zum Ende der Lebenszeit, wie in Kapitel 2.3.3 vorgestellt, ist aufgrund des Speicherbedarfs zu vermeiden. Stattdessen wird serverseitig eine *MFA*-Nonce zum *MFA*-Versuch vermerkt, welche durch Integration an den jeweiligen *MFA*-Token gekoppelt wird. Nach Vorzeigen eines Faktors wird unabhängig vom Erfolg der Iteration die Nonce geändert. Ältere Tokens des *MFA*-Versuchs sind damit automatisch invalidiert. Ein Angreifer ist an die vorgegebene Anzahl eingeräumter Fehlversuche gebunden, da er ältere Tokens mit niedrigerem Fehlerzähler nicht erneut verwenden kann. Indem serverseitig nur eine *MFA*-Nonce je Account vermerkt wird, entsteht ein *Mutex*, der lediglich einen parallelen *MFA*-Versuch erlaubt.

Bei Abarbeitung einer *MFA*-Anfrage sollte der Server die Nonce so früh wie möglich ändern, nachdem er den eingereichten *MFA*-Token validiert hat. Andernfalls könnte ein Angreifer unter Umgehung des Rate-Limitings viele zeitlich annähernd parallele Anfragen mit einem *MFA*-Token und verschiedenen Nachweiswerten stellen. Alle Anfragen bis zur ersten Nonce-Änderung wären gültig. Zur Mitigation dieses *Time-Of-Check to Time-Of-Use (TOCTOU)*-Problems sollte der Server deshalb im letzten Schritt vor Beantwortung eines *MFA*-Requests die Nonce nochmals prüfen. Stimmt diese nicht mehr mit der Nonce zu Beginn der Request-Abarbeitung überein, muss ein Fehler ausgelöst werden.

Die Implementierung sieht vor, dass jedem Faktor ein Nickname zugeordnet wird, damit der Nutzer sie in der *UI* zu den *MFA*-Einstellungen unterscheiden kann. Diese Nicknames dürfen nicht im Prozess zur Authentifizierung verwendet werden. Ein Angreifer könnte sonst aus dieser Information Rückschlüsse darüber ziehen, an welchem Ort und in welcher Gestalt weitere Faktoren zu finden wären. Deshalb sollten auch Größe und Inhalt des Pools gespeicherter Faktoren nicht preisgegeben werden. Das zu erzielende *MFA*-Level ist ebenfalls geheim zu halten. Aus Gründen der *UX* kann jedoch die Liste möglicher Kategorien zur Wahl des nächsten Faktors bekannt gemacht werden. Diese Kategorien ergeben sich aus den in diesem Kapitel vorgestellten Verfahren.

4.1.3 Mehr-Augen-Prinzip

Durch eine iterative *MFA* kann die Sicherheit des Prozesses zur Authentifizierung gesteigert werden. Allerdings verbessert sich die Sicherheit des Verfahrens mit Erhöhung des *MFA*-Levels nur bis zu einem bestimmten Punkt. Ein Nutzer muss die notwendige Menge an Faktoren in seinem Zugriffsbereich halten, um sich authentifizieren zu können. Bei einem Hochsicherheitsverfahren ist es für einen Angreifer folglich einfacher gegen den Nutzer vorzugehen als gegen das Verfahren selbst. Ein Angreifer könnte z. B. durch Anwendung von Social Engineering, Erpressung oder Folter Zugang erlangen.
Deshalb kann die Verantwortung zur Autorisierung einer Aktion auf mehrere Personen verteilt werden, was als Mehr-Augen-Prinzip bezeichnet wird. Die zu erfüllenden Aktionen müssen vordefiniert sein oder sich aus dem System ergeben, z. B. Bestätigung jeder Finanz-Transaktion durch den ausführenden Mitarbeiter und den Vorgesetzten. Zu jeder möglichen Aktion ist serverseitig zu vermerken:

- Inhalt der durchzuführenden Aktion
- Nutzerkennungen der zur Durchführung berechtigen Accounts
- Anzahl an zur Durchführung notwendigen Nutzern

- Eingeräumte Zeitspanne zur Einholung der Zustimmung von den Nutzern

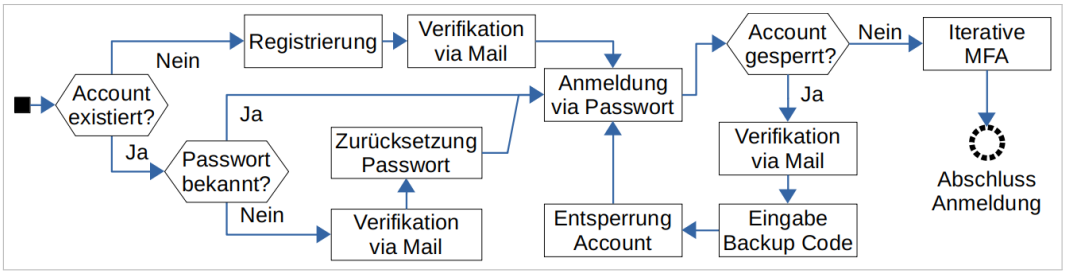
Das Token-Konzept zur iterativen *MFA* kann um ein iteratives Mehr-Augen-Prinzip erweitert werden. Nach erfolgreicher *MFA* vorautorisiert der anfordernde Nutzer die betreffende Aktion. Der Server erzeugt einen Token zum Vorgang der Autorisierung. Er wird über einen sicheren Kanal an einen der weiteren berechtigten Nutzer übermittelt. Daraufhin durchläuft dieser Nutzer die iterative *MFA*, um seine Zustimmung zur Aktion zu bezeugen. Der Server stellt daraufhin einen neuen Token mit den aktualisierten Vorgangsdaten aus. Sobald die notwendige Anzahl berechtigter Nutzer innerhalb des Zeitfensters zugestimmt hat, ist der Vorgang vollautorisiert.
Das Verfahren kann neben Hochsicherheitsanwendungen auch bei Nutzern mit fehlender Geschäftsfähigkeit eingesetzt werden, z. B. wenn Eltern ihrem Kind einen Account bereitstellen.

4.1.4 Ergänzungen zum Referenz-Protokoll

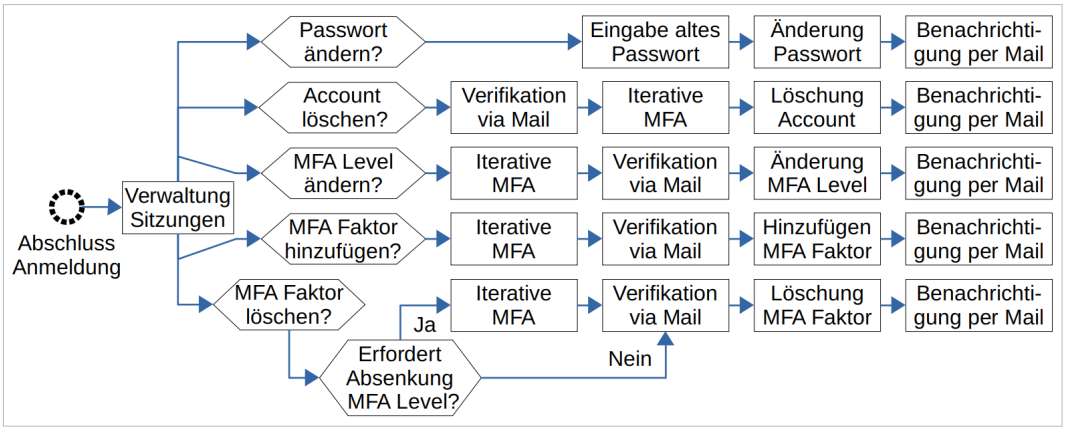
Nachfolgend werden die Änderungen am Referenz-Protokoll aus Kapitel 2.3 zur Unterstützung der in Kapitel 4.1.2 vorgestellten, iterativen *MFA* erläutert. Da die *MFA* optional sein soll, bleibt die Registrierung mit Passwort zur *1FA* unverändert. Jedoch sollten Nutzer zur Einrichtung einer *MFA* animiert werden, um die systemische Sicherheit zu steigern. Zur Anmeldung gibt der Nutzer zunächst sein Passwort an und durchläuft dann die iterative *MFA*. Abbildung 4.2 zeigt eine mögliche Integration der *MFA*, die nun erläutert wird.

Bei Passwortverlust erfolgt die Zurücksetzung, wie in Kapitel 2.3.4 dargelegt, nur über die Mail-Verifikation. Damit bleibt der Zugriff auf den Mail-Account weiterhin gleichbedeutend mit dem Besitz des Faktors Passwort. Die Ergänzung einer *MFA* wäre nicht sinnvoll, da das Passwort als Faktor lediglich ausgetauscht wird. Auf die iterative *MFA* hat dies keinen Einfluss. Deshalb ist im Kontext einer Session auch zur Passwortänderung weiterhin die Angabe von altem und neuem Passwort ausreichend, vergleiche Kapitel 2.3.5.
Zur Löschung eines Accounts wird, wie in Kapitel 2.3.6 erläutert, zunächst eine Löschanfrage per Mail versendet. Erst nach dieser Mail-Verifikation wird die iterative *MFA* angewendet, um die Löschbarkeit des Accounts nicht vom Zugriff auf das zugehörige E-Mail-Konto abhängig zu machen.

Durch die *MFA* kommen Protokollfunktionen hinzu: Änderung des *MFA*-Levels sowie Hinzufügen und Löschen von *MFA*-Faktoren. Die Aktionen sind besonders schützenswert: Durch Änderung des *MFA*-Levels kann ein Angreifer die *MFA* gänzlich umgehen. Durch Hinzufügen eigener *MFA*-Faktoren kann ein Angreifer



(a) Protokollfunktionen ohne Sitzung



(b) Protokollfunktionen mit Sitzung

Abbildung 4.2: Mögliche Ergänzung einer MFA zum Referenz-Protokoll
Quelle: Eigene Darstellung

die *MFA* ebenfalls überwinden. Das Löschen von Faktoren ist ebenfalls riskant. Stehen durch Löschung weniger Faktoren als benötigt zur Verfügung, so muss das *MFA*-Level gesenkt werden, was ein Angreifer für sich nutzen kann. Deshalb wird die Durchführung dieser Aktionen im Kontext einer Session zusätzlich abgesichert: Nach iterativer *MFA* sind die Änderungen als unverifiziert hinterlegt. Erst durch die Mail-Verifikation werden sie aktiv und finden Anwendung.

Zu einem Account sollten mehr *MFA*-Faktoren hinterlegt sein, als zur Authentifizierung notwendig sind. Andernfalls würde der Nutzer bei Verlust eines Merkmales den Zugang verlieren. Das verknüpfte Mail-Konto ist nicht als Backup-Faktor verwendbar, da es bereits zur Passwort-Zurücksetzung dient. Sonst würde es als Äquivalent zu zwei Faktoren fungieren. Hiergegen eine Exklusiv-Oder-Regel anzuwenden, würde den Account-Zugriff bei Verlust beider Faktoren verhindern.

Stattdessen wird ein Backup-Code eingeführt, der sich als zusätzlicher *MFA*-Faktor einsetzen lässt. Hierbei handelt es sich um ein zufällig gewähltes, möglichst langes Passwort. Die Maßnahmen der Passwortverarbeitung sind damit übertragbar. Der Backup-Code wird einmalig erzeugt und angezeigt, wenn der Nutzer im Kontext einer Session erstmalig von einer *1FA* zur *MFA* wechselt. Dazu konfiguriert der Nutzer seine *MFA*-Faktoren an der Anwendung. Jedoch besteht ein

psychologisches Problem: Die Erwartungshaltung eines Nutzers, nun seine *MFA*-Faktoren zu registrieren, deckt sich nicht mit der Anforderung, zunächst einen Backup-Code zu notieren. Nutzer werden dazu neigen, diesen Schritt zu überspringen. Zulasten der *UX* sollte der Backup-Code deshalb initial abgefragt werden, bevor eine *MFA* konfiguriert werden darf. Abbildung 4.3 fasst den Vorgang zusammen. Der Backup-Code kann optional auch zur Entsperrung eines Accounts mit zu vielen *MFA*-Fehlversuchen dienen, siehe Abbildung 4.2a.
Alternativ zur Einführung von Backup-Codes ist für Organisationen mit geschlossenem Nutzerkreis auch ein manuelles Verfahren implementierbar: Bei Account-Sperrung oder Faktor-Verlust authentifiziert der betroffene Nutzer sich gegenüber einem Mitarbeiter der zuständigen Abteilung, welcher die Aktion vornimmt.

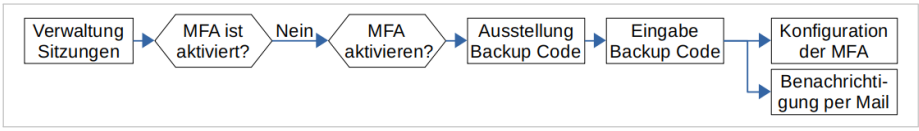


Abbildung 4.3: Ablauf zur Aktivierung einer MFA zu einem Account
Quelle: Eigene Darstellung

In diesem Abschnitt wurden die notwendigen Schritte zur Integration einer ergänzenden, optionalen, iterativen *MFA* dargelegt. Nachfolgend werden verschiedene Verfahren erläutert, die hierzu als *MFA*-Faktoren nutzbar sind.

4.2 Symmetrische Verfahren

In diesem Abschnitt werden *MFA*-Verfahren vorgestellt, bei denen Client und Server ein gemeinsames Geheimnis teilen. Beide Seiten verfügen über das Geheimnis in unterschiedlichen Verarbeitungsstufen, z. B. Klartext oder Hashwert. Damit sind die Geheimnisse als äquivalent zu Passwörtern zu betrachten, welche in abgewandelten Formen verwendet werden. In Abgrenzung dazu werden in Kapitel 4.3 Verfahren betrachtet, die als kryptografisches Primitiv asymmetrische Schlüssel anstatt Hashing verwenden.

4.2.1 Clientseitige Passwort-Generatoren

Vor Untersuchung von Verfahren zur iterativen *MFA* werden zunächst aufbauend auf Kapitel 4.1.1 Merkmale und Einsatzmöglichkeiten clientseitiger *MFA* erläutert. Diese sind in Kombination mit der iterativen *MFA* anwendbar.
Clientseitige Passwort-Generatoren erlauben eine webseiten- und applikationsübergreifende Verwaltung von Passwort-Besitz mit hoher Entropie durch den *UA* oder angebundene Software. Dabei können die erzeugten Passwörter aus einem Main-Secret abgeleitet oder individuell für jeden Account gespeichert werden.

Beide Varianten stellen bei korrekter Umsetzung eine Abwandlung zum clientseitigen *MFA*-Schema aus Kapitel 4.1.1 dar und werden nachfolgend betrachtet.

Ableitung von Passwörtern aus einem Main-Secret

Dieses Verfahren ist für Webapplikationen einsetzbar, z. B. als Browser-Extension. Zur Passwort-Erzeugung wird der Domainname *domain* einer aufgerufenen Webseite als Salt für das *Hashing* mit dem Main-Secret *secret* genutzt:
$pw := hash_{cli}(hmac_{cli}(secret, domain), cost_{cli})$
Das Verfahren ist portabel, da nur *secret* vorzuhalten und zu handhaben ist. In [RJM+05] werden mögliche Variationen des Verfahrens vorgestellt.

Wenn das Main-Secret sich aus Passwortkenntnis ergibt, findet das Verfahren äquivalent zum Schema aus Kapitel 3.2.2 statt. Daraus leiten sich die Anforderungen an ein langsames Hashing ab. Einzig der Erzeugungskontext verlagert sich von der Webapplikation auf den *UA*.
Das Problem des Verfahrens besteht in dem vorausgesetzten Vertrauen zu allen einbezogenen Domains: Sei ein Dienstbetreiber gegeben, der kein hinreichend langsames Server-Hashing verwendet und von einem *Data Breach* betroffen ist. Ein Angreifer nimmt an, dass Nutzer das Verfahren mithilfe eines bestimmten Tools bewerkstelligen. Durch Ausführung von vermuteter Client- und Server-Hashfunktion kann der Angreifer versuchen, das Main-Passwort zu bestimmen. Hat er es gefunden, ist ein Credential-Stuffing für andere Websites trivial möglich.[LJR17] Wie in Kapitel 3.2.1 gezeigt, ist ein Angriff auf das Main-Secret mit niedriger Entropie unter Ausführung beider Hashfunktionen effektiver als ein Angriff auf das erzeugte Passwort mit hoher Entropie. Das Verfahren führt Schwachstellen bezüglich der Data-Breach-Security systematisch herbei.

Alternativ könnte das Main-Secret sich aus Passwortbesitz mit hoher Entropie ergeben. Wird das Main-Secret mittels Passwort (als Passwortkenntnis) lokal verschlüsselt vorgehalten, liegt eine starke Client-*2FA* vor. Konzeptionell besteht jedoch ein weiteres Problem des Verfahrens in der mangelhaften Handhabung einer Passwortänderung: Ein erzeugtes Passwort kann bekannt oder der Dienst kompromittiert werden. Das betreffende Passwort ist dann vom Verfahren entkoppelt zu wechseln. Ansonsten sind das Main-Secret und damit alle daraus erzeugten Passwörter zu ändern. Damit schwindet der Vorteil gegenüber individuellen Secrets je Applikation.

Erzeugung individueller Passwörter je Applikation

Dieser Ansatz entspricht der Arbeitsweise eines Passwort-Managers, wobei diese Funktionalität direkt in den *UA* eingebettet sein kann. Beispielsweise verfügen moderne Webbrowser in der Regel über die Funktion, Passwörter mittels *CSPRNG* zu erzeugen, zu einer Domain abzuspeichern und bei Webseitenaufruf automatisch einzufügen.[Hod21] Mit der „Credential Management API“ befindet sich zudem eine Schnittstelle für Webanwendungen im Standardisierungsprozess, über die Webseiten im *UA* Passwörter hinterlegen und zur Webseite gespeicherte Passwörter abrufen können sollen.[Hod21]
Der Speicher für den Passwortbesitz sollte durch symmetrische Verschlüsselung geschützt werden, wozu ein Main-Passwort verwendet werden kann. Damit liegt eine starke Client-*2FA* vor. Die Länge erzeugter Passwörter kann so gewählt werden, dass sie ein hohes Sicherheitsniveau bieten, aber auch mit vertretbarem Aufwand durch den Nutzer einmalig an weiteren Geräten eingegeben werden können, z. B. 16 zufällige Zeichen. Zu lange Zufallspasswörter motivieren den Nutzer zur Übertragung über einen selbstgewählten Kanal, der potenziell unsicher ist.

Der Ansatz ergänzt die Passwortkenntnis um Passwortbesitz und bietet dabei eine gute Kombination aus *UX* und Passwort-Sicherheit. Eine *MFA* muss durch den Dienstanbieter nicht bereitgestellt werden, siehe Kapitel 4.1.1. Allerdings bleiben die Probleme einer rein passwortbasierten Authentifizierung bestehen. Um ein Passwort zur Laufzeit des *UA* einfügen zu können, muss es entschlüsselt vorliegen. In Abhängigkeit von Zeitpunkt und Verweildauer im unverschlüsselten Zustand kann ein Angreifer möglicherweise Passwörter auch ohne Nutzereingabe am Client abgreifen. Ansonsten reicht einem Angreifer weiterhin die einmalige Einblicknahme des Clients während einer Authentifizierung, da Passwortbesitz und -kenntnis am gleichen Gerät erbracht werden. Der Einsatz von *One Time Password (OTP)*-Verfahren mitigiert dieses Problem.

Clientseitige Passwort-Generatoren stellen eine nützliche Verstärkung der Passwort-Authentifizierung dar. Da sie außerhalb des Applikationskontextes agieren, beeinträchtigen sie die weiteren Maßnahmen zur iterativen *MFA* nicht.

4.2.2 One Time Passwords (OTP)

Ein *One Time Password (OTP)* ist einmalig verwendbarer Passwortbesitz. Dadurch werden Eavesdropping und Replay-Attacken vermieden. Eine Anmeldung kann damit auch an einem eingeschränkt vertrauenswürdigen Gerät durchgeführt werden, ohne sämtliche Secrets preiszugeben. Voraussetzung ist eine Erzeugung der *OTPs* durch ein zweites, vertrauenswürdiges Gerät.

Klassifikation

OTP-Verfahren lassen sich in statische und dynamische unterteilen. Bei statischen Verfahren wird der Pool verwendbarer *OTPs* einmalig befüllt, z. B. als ausgedruckte Kennwortliste. Bei jeder Authentifizierung ist dann das gewählte Element aus dem Pool zu entfernen. Wenn der Pool leer ist, muss er neu befüllt werden. Das Verfahren kann entweder mit wahlfreiem Zugriff auf ein *OTP* im Pool oder durch sequenzielle Abarbeitung einer *OTP*-Liste erfolgen. Der sequenzielle Ansatz ist vorzuziehen, da dann nur ein *OTP* zu einem Zeitpunkt Gültigkeit besitzt. Der Einsatz eines vorbefüllten Pools weist nicht nur das Nachfüllproblem auf. Er hat außerdem die Schwäche, dass eine einmalige Einblicknahme des Pools genügt, um zukünftige Authentifizierungen zu absolvieren. Als Gegenmaßnahme werden die *OTPs* dynamisch generiert, sodass eine Herausgabe nur für das jeweils aktuelle *OTP* erfolgt. Ein *OTP*-Generator kann ereignisgesteuert, zeitgesteuert oder mittels Challenge-Response-Verfahren als Push-*OTP* konstruiert werden.[Poh19] Es folgt die Vorstellung dieser drei Varianten.

Push OTP

Challenge-Response-Verfahren als Technik zum Zugriffsnachweis für einen Transportkanal wurden in Kapitel 3.3.1 eingeführt. Im Einsatz als Push-*OTP* erhält der Nutzer eine Nachricht mit dem *OTP* zugesandt und tippt es ab. Nachteilig sind die Zeitverzögerung zum Beziehen des *OTP* durch die Netzwerk-Übertragung sowie die Voraussetzung eines Netzwerkzugangs am Zweitgerät. Außerdem muss das Zweitgerät zum *OTP*-Empfang über einen sicheren Kanal erreichbar sein. Dies ist durch eine separate Applikation oder ein Protokoll mit generischen *UAs* erzielbar, wozu die Bereitstellungsformen *SMS* und E-Mail zählen.
Beim *SMS*-Verfahren wird eine Gerätetrennung erzielt, wenn der Nutzer sich am Computer anmeldet und am Mobilgerät das *OTP* empfängt. Bei Anmeldung am Mobilgerät entfällt diese Trennung. Dafür ist in diesem Fall eine *UX*-Verbesserung möglich: Der *UA* kann den *OTP* dann automatisch aus der *SMS* in die Webapplikation einfügen. Eine Implementierung hierzu ist WebOTP, dessen Standardisierung in [Got21] begonnen wurde. Abbildung 4.4 zeigt eine passende Beispiel-*SMS*. Für das Verfahren ist von dieser nur die letzte Zeile relevant. Neben dem *OTP* enthält sie den Domainnamen der Webapplikation. Das Einsetzen des *OTPs* durch den *UA* erfolgt nur, wenn der Domainname mit dem der aufgerufenen *URL* übereinstimmt. Für Nutzer ist es jedoch durch manuelle Codeeingabe leicht sich trotzdem darüber hinwegzusetzen. Es ist sehr wahrscheinlich, dass dies passiert: Die Erwartungshaltung der Nutzer kann von einer manuellen Eingabe ausgehen.

Ansonsten kann auch eine technische Fehlfunktion angenommen werden. Deshalb stellt das Verfahren vorrangig eine *UX*-Optimierung und nur begrenzt eine Sicherheitsmaßnahme dar. Dies wird in der Sicherheitsbetrachtung am Ende des Abschnitts ausführlich thematisiert.

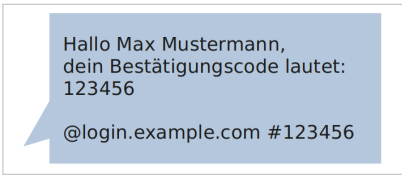


Abbildung 4.4: Beispiel einer SMS mit OTP im WebOTP-Format
Quelle: Eigene Darstellung

Bei den generischen Verfahren *SMS* sowie E-Mail erfolgt der Nachrichtentransport unverschlüsselt oder transportverschlüsselt, aber standardmäßig ohne *End-To-End Encryption (E2EE)*. Ein Angreifer kann folglich die Authentifizierung via Push-*OTP* umgehen, indem er sich als *MITM* in den Transportkanal einbringt. Dagegen ist eine starke *E2EE* zu implementieren. Dies könnte z. B. durch *GNU Privacy Guard (GPG)*-geschützte E-Mails oder den Messenger Signal erfolgen. Diese Lösungen sind weniger generisch. Die resultierenden Anforderungen an Nutzer und Endgerät schränken die Deployability ein. Stattdessen kann eine eigenständige *OTP*-App eingesetzt werden, die Nachrichten per *TLS* vom Dienstanbieter empfängt. In diesem Aufbau fungiert *TLS* zur *E2EE* zwischen Nutzer und Dienstanbieter. Da diese spezialisierte *OTP*-App allerdings nur für begrenzt viele Betriebssystem-Plattformen und -Versionen bereitstellbar ist, wird die Deployability dennoch eingeschränkt. Außerdem schadet es der *UX*, wenn Nutzer für jede Anwendung eine separate Login-App installieren müssen.

HMAC-based OTPs (HOTP)

HOTP ist ein ereignisgesteuertes Verfahren. Das Zweitgerät erzeugt den aktuellen *HOTP*-Wert. Nach Eingabe am Erstgerät erfolgt die Übermittlung an den Server. Der Server generiert ebenfalls den aktuellen *HOTP*-Wert und vergleicht ihn mit dem vorgelegten. Dazu müssen Zweitgerät und Server einen symmetrischen Schlüssel *key* teilen, welcher auch als Seed bezeichnet wird. Um eine Wertesequenz erzeugen zu können, ist auch eine Nonce zu teilen, die das *HOTP*-Verfahren als Zähler *ctr* umsetzt. Die Anforderung einer Authentifizierung stellt das Ereignis dar, bei dem der Zähler zu inkrementieren ist. Ein *HOTP*-Wert wird als Ziffernfolge der Länge *digits* repräsentiert. Die Spezifikation [MHN+05] fordert eine Mindestlänge von sechs Ziffern, woraus sich 10^6 mögliche *OTP*-Werte ergeben. Ein *HOTP*-Wert wird gebildet als:

$$hotp(key, ctr) = truncate(hmac(ctr, key), digits)$$

Die Funktion *truncate* bildet die Ausgabe-Bitfolge von *hmac* auf *digits* Ziffern ab. In [MHN+05] werden Maßnahmen zur Verhinderung einer Verzerrung der gleichverteilten Hashwerte durch diese Zuordnung erläutert. Die Spezifikation [MHN+05] sieht lediglich SHA-1 zur *HMAC*-Bildung vor, wobei das Verfahren nicht an diese Hashing-Funktion gebunden ist.
Die Zähler von Zweitgerät und Server können sich desynchronisieren: Der Server erhöht den Zähler nur bei Anforderung einer Authentifizierung und der Client bei jeder Interaktion durch den Nutzer. Deshalb sollte der Server neben dem *HOTP* für den aktuellen Zählerwert auch eine begrenzte Zahl nachfolgender *HOTPs* akzeptieren. Mit der Größe dieses Look-Ahead-Windows steigt aber auch die Wahrscheinlichkeit, einen ungültigen Wert zu akzeptieren. Wenn der vorgelegte Wert nicht im Look-Ahead-Window enthalten ist, muss eine Resynchronisation gestartet werden. Dazu fordert der Server eine gewisse Anzahl aufeinanderfolgender *HOTP*-Werte an, um den Zählersprung authentifizieren zu können. [MHN+05]

Der Werteraum besitzt eine Größe von 10^{digits} mit $digits \geq 6$ aber mit *digits* aufgrund der *UX* so klein wie möglich. Dieser winzige Werteraum prädestiniert das Verfahren für Brute-Force-Angriffe. Deshalb sollte es als nicht-primärer Faktor in einer iterativen *MFA* mit begrenzten Fehlversuchen implementiert werden, siehe Kapitel 4.1.2. Alternativ ist, wie in [MHN+05] empfohlen, ein Rate-Limiting als Server-Throttling einsetzbar, sodass der Faktor erst nach einer Cool-Down-Zeit erneut verwendet werden kann.

Die Sicherheit des Verfahrens basiert auf der Geheimhaltung des Seeds. Im Backend muss der Seed im Klartext vorliegen, um *OTPs* erzeugen zu können. Zum Schutz lässt sich lediglich eine symmetrische Verschlüsselung der Seeds mit Schlüsselspeicherung außerhalb der *DB* anwenden. Dieser Ansatz ist damit zur Geheimhaltung mittels Peppering äquivalent, vergleiche Kapitel 2.2.3 sowie 3.3.3. Das Schutzniveau fällt aufgrund des fehlenden Hashings dennoch geringer als bei regulären Passwörtern aus. Damit sichert das Verfahren nur reguläre Zugriffe mit begrenztem Schutz vor irregulären ab.
Auf Seiten des Nutzers ist der Seed ebenfalls zu schützen. Dazu ist als Zweitgerät dedizierte Hardware verwendbar, die keine zerstörungsfreie Extraktion des Seeds erlaubt. Wenn der Seed durch den Hersteller auf der Hardware aufgebracht wird, besteht jedoch die Gefahr der Erstellung eines Zweitschlüssels, indem der Hersteller eine Kopie des Seeds speichert. Dies kann auch absichtlich geschehen, z. B. als Backup-Komfortfunktion für den Fall des Geräteverlustes. Der Angriffsvektor wird dadurch allerdings auf den Hersteller ausgeweitet, was in der Vergangenheit

bereits aktiv ausgenutzt wurde, siehe [Bac11]. Zum Schutz des Seeds sollte eine lokale Passwort-Verschlüsselung für das Zweitgerät etabliert werden, z. B. durch Eingabe einer *Personal Identification Number (PIN)* am Zweitgerät.

Das Verfahren erlaubt auch eine wechselseitige Authentifizierung: Dabei legt zuerst der Client das aktuelle *OTP* am Server vor. Nach Prüfung generiert der Server das nächste *OTP* und übermittelt es an den Client. Dieser inkrementiert den Zähler und kann das *OTP* verifizieren. [MHN+05]

Time-based OTPs (TOTP)

TOTP ist ein zeitgesteuertes Verfahren. Der ereignisgesteuerte Zähler des *HOTP*-Verfahrens wird durch einen zeitgesteuerten ersetzt. Dies macht den Zähler zustandslos und mitigiert das Synchronisationsproblem für Nutzer. Als Zeitbasis dient die *Epoch*, wobei eine Festlegung des Epoch-Startwerts auch abweichend von 0 möglich ist. Eine Zeitperiode der Länge *period* wird gewählt, typischerweise 30 Sekunden [MRPM11]. Mit dem momentanen Epochwert t_{now} werden aktueller Zählerwert und *TOTP*-Wert berechnet als:

$$ctr := \left\lfloor \frac{t_{now}}{period} \right\rfloor$$
$$totp(key, ctr) = truncate(hmac(ctr, key), digits)$$

Das Synchronisationsproblem der Zähler verlagert sich auf die Zeitgeber der Systeme, was die Usability verbessert. Wenn die Abweichung der Zeitgeber ansteigend ist, kann dies eine langfristige Verwendung des Faktors verhindern. Der Zählerwert kann sich zwischen Eingabe durch den Nutzer und Empfang durch den Server geändert haben. Deshalb muss eine Policy gewählt werden, um ein größeres Zeitfenster aufzuspannen, sodass z. B. auch der vorhergehende und nachfolgende Zählerwert akzeptiert werden.
Die Sicherheitsbetrachtung zu *HOTP* ist übertragbar. Zusätzlich besteht das Problem einer Verletzung der Nonce-Eigenschaft, wenn ein Angreifer einen abgefangenen *TOTP*-Code im Zeitfenster nochmals einlösen kann. Deshalb ist die Server-Implementierung einer Cool-Down-Zeit zum nächsten Zählerwert für *TOTPs* obligatorisch. Dazu wird im Backend je Account der zuletzt verwendete Zählerwert gespeichert. Zur Zustandsreduzierung ist die Angabe löschbar, sobald der zuletzt verwendete nicht mehr im Zeitfenster des aktuellen Zählerwerts liegt.

Zur Registrierung eines *TOTP*-Faktors zu einem Account muss der Server einen Seed erzeugen und mit den Parametern (eingesetzter Hashalgorithmus, *period*, *digits*) an den Client übermitteln. Folgende Varianten sind hierzu möglich:

- Variante 1: Der Nutzer meldet sich am Erstgerät an, empfängt dort die Parameter und überträgt sie händisch auf das Zweitgerät.
- Variante 2: Der Nutzer meldet sich mit seinen Credentials am Zweitgerät an und erhält dort direkt die Parameter, sodass eine Übertragung entfällt.

Für Variante 1 muss ein Datenformat für die Übertragung auf das Zweitgerät festgelegt werden. Das Unternehmen Google hat hierzu ein spezielles *URI*-Schema vorgeschlagen.[1] Da das Format auch von anderen *OTP*-Apps als dem Google Authenticator übernommen wurde[2], fungiert es als Quasi-Standard. Diese Provisionierungs-*URI* kann mittels *QR*-Code-Scan oder als Fallback durch manuelle Eingabe auf das Zweitgerät transportiert werden. Auf dem Zweitgerät ist eine generische *OTP*-App verwendbar, die *OTPs* für mehrere Applikationen bereitstellt. Die App sollte *OTP*-Seeds zum Schutz nur nach Anwendung einer lokalen, symmetrischen Passwort-Verschlüsselung persistieren.

Variante 2 wurde in [UY14] als *Time-based OTP Authentication via Secure Tunnel (TOAST)* vorgeschlagen. Ein Transport zum Zweitgerät entfällt, da der Nutzer sich dort zur Faktor-Registrierung anmeldet. Das Ablesen und Übertragen eines *OTP* auf das Erstgerät erfolgt für *TOAST* wie bei Variante 1.
Alternativ zu *TOAST* könnte das Zweitgerät den *OTP* auch direkt an den Server übertragen. Das Ablesen von *OTPs* würde entfallen und die *UX* wäre gesteigert. Allerdings wird die iterative *MFA* dann in *1FA*-Vorgänge aufgeteilt. Da Erst- und Zweitgerät sich nicht im gleichen Netzwerk befinden müssen, kann der Server die Vorgänge nicht gruppieren, auch nicht über die Request-*IP*-Adresse. Die Vorteile einer iterativen *MFA* gehen verloren, siehe Kapitel 4.1.2. Ein Angreifer, der das Passwort mittels Anfragen erraten will, wird auf einen Versuch je *period* beschränkt. Jedoch ermöglicht dies, einen legitimen Nutzer auszusperren: Der Angreifer legt je *period* einen beliebigen *OTP* für den Account vor, sodass der Nutzer nicht mehr den richtigen vorzeigen kann. Deshalb ist lediglich *TOAST* als Schema für Variante 2 geeignet.

Jedoch wird Variante 1 präferiert: Die niedrigeren Anforderungen an das Zweitgerät erhöhen die Deployability. Die geringere Kopplung zur Applikation erlaubt die Nutzung einer generischen *OTP*-App für mehrere Anwendungen. Dadurch wird die Usability erhöht und die Security-Betrachtung auf wenige, zentralisierte Apps fokussiert.

[1] Zur Dokumentation des *OTP-URI*-Schemas siehe `https://github.com/google/google-authenticator/wiki/Key-Uri-Format`.
[2] Siehe z. B. `https://freeotp.github.io/qrcode.html` für die freie App „FreeOTP Authenticator“.

Sicherheitsbetrachtung

OTP-Verfahren stellen minimale Anforderungen an den *UA*, da lediglich eine Ziffereneingabe als *UI* benötigt wird. Die Abbildung des Verfahrens kann vollständig durch die Applikation erfolgen, nicht durch den *UA*.
Allerdings ermöglicht der Einsatz von Webbrowsern Phishing-Angriffe gegen *OTP*-Verfahren: Da ein *OTP* nicht an eine Domain gebunden ist, besteht das Problem der Mittelbarkeit. Im Gegensatz dazu kann z. B. ein clientseitiger Passwort-Generator so gebaut werden, dass der *UA* das Passwort nur bei Aufruf der korrekten, zugehörigen Domain einfügt, siehe Kapitel 4.2.1.
Im Kontext von Webapplikationen wird anstatt der Domain verfeinernd die aufgerufene Origin betrachtet. Diese besteht als Untermenge zum *URL* aus Protokoll, Domain inkl. Subdomains und Server-Port. Für `https://example.org` stellen beispielsweise die folgenden *URLs* abweichende Origins dar:

- `https://example.com`
- `https://example.org:4443`
- `https://sub.example.org`
- `http://example.org`

Die *OTP*-Eingabe erfolgt auf der durch den Nutzer aufgerufenen Webseite, also potenziell auch auf einer Phishing-Webseite. Sei ein Nutzer mit Account bei `meine-bank.de` gegeben. Ein Angreifer könnte mittels Social-Engineering Phishing-Mails verschicken, die Links zu Scam-Webseiten enthalten, wie z. B.:

- `meine-bank.com`, `meine.bank.de`, `meine-bank-online.de`
- `rneine-bank.de`, `meine-bahk.de` usw.

Gibt der Nutzer seine Zugangsdaten auf einer angreiferkontrollierten Webseite ein, kann dieser sie als *MITM* an die echte Webseite weiterleiten, um die Authentifizierung auszuführen. Dieser Angriff mit ähnlich aussehenden Domains wird als homografisches Phishing bezeichnet.
Für Mobilgeräte mit begrenzter Bildschirmfläche kann der Angriff auch mit sehr langen Domainnamen erfolgen, die dann in der Darstellung abgeschnitten werden, z. B. `anmeldung.meine-bank.de.attacker.com`.
Der Einsatz von Punycode-Domains kann eine visuelle Erkennung von homografischem Phishing unmöglich machen: Punycode-Domainnamen erlauben neben dem *American Standard Code for Information Interchange (ASCII)*-Zeichensatz auch die Verwendung encodierter Unicode-Zeichen. Dem *UA* steht es frei, die Punycodes als Unicode-Zeichen darzustellen. Der kyrillische Buchstabe „а“ (U+0430) lässt sich dann z. B. optisch nicht mehr vom lateinischen „a“ (U+0061) unterscheiden. Hierdurch können unterschiedliche Domains gebildet werden, die

optisch gleich sind, wie `bank.de` und `bank.de`.
Zudem ist ein Typosquatting-Angriff möglich, bei dem der Angreifer erwartet, dass der Nutzer sich bei der *URL*-Eingabe vertippt. Hierzu kann er, dem Beispiel folgend, Domains registrieren wie `miene-bank.de` oder `meine-bamk.de`.
Außerdem sind weitere browser-spezifische Angriffe möglich: So könnte eine Phishing-Seite zur Nutzer-Täuschung z. B. den Browser in den Vollbildmodus versetzen und dann Browser-*UI* inklusive abgeänderter *URL*-Zeile nachempfinden.

Der Einsatz von browserbasierten *OTP*-Verfahren impliziert damit eine Origin-Prüfung durch den Nutzer. Wie dargelegt ist diese aufwändig bis unmöglich. Der Einsatz von *OTPs* bietet in dieser Hinsicht keinerlei Vorteil gegenüber einer Passworteingabe, die gleichermaßen anfällig ist. Deshalb sollte zur browserbasierten *MFA* mindestens ein Faktor mit Origin-Abhängigkeit eingesetzt werden. Entsprechende Verfahren werden später in diesem Kapitel erläutert. Für eine browserbasierte *2FA* mit Passwort als erstem Faktor sollten *OTPs* daher, wenn überhaupt, nur als Fallback zum Einsatz kommen.
Anzumerken ist, dass dieses Problem spezifisch für Webbrowser ist. Ein *UA*, der die Applikationslogik als Desktop-Programm inkludiert, wäre nicht direkt betroffen. Der Nutzer müsste zunächst die Software inklusive Trojaner, z. B. als Update getarnt, von einer bösartigen Webseite herunterladen und installieren. Das Phishing-Problem besteht dann in der Wahl von Softwarequellen und nicht in den Authentifizierungsverfahren der Applikation.

4.2.3 Symmetrische Device-Cookies

Ein Device-Cookie stellt ein Merkmal dar, welches durch die Applikation auf dem Client-Gerät aufgebracht wird und der User-Agent-Bindung dient. Hierzu wird ein geheimer Wert im *UA* gespeichert, welcher seine Wiedererkennung ermöglicht. Der *UA* fungiert damit als zusätzlicher Faktor zur Authentifizierung. Im Gegensatz zu den anderen *MFA*-Verfahren wird das Merkmal nicht durch den Nutzer bereitgestellt, sondern ergibt sich aus dem Protokoll-Aufbau des Systems. Die Technik ist für mehrere Zwecke nutzbar, wie nachfolgend erläutert.

Einsatz zum Rate-Limiting

Der Begriff „Device-Cookie“ wurde im Zusammenhang mit Rate-Limiting-Ansätzen geprägt, welche Gegenstand von Kapitel 5.1.5 sind.[3] Ein Rate-Limiting kann Anmeldeversuche, mit dem Ziel Passwörter zu erraten, unterbinden. Diese Brute-

[3] Als Technologie für Device-Cookies müssen keine Cookies eingesetzt werden. Der Name spielt lediglich auf die Speicherung im Client und den Austausch mit dem Server an.

Force-Angriffe können sich gegen einen oder mehrere Accounts richten und werden verhindert, indem der Server entsprechende Anfragen abweist. Die Gefahr dabei ist, nicht nur den Angreifer, sondern auch legitime Nutzer vom System auszuschließen. Es gilt legitime Anmeldeversuche von illegitimen zu unterscheiden. Die korrekte Passwortangabe ist als Merkmal hierzu nicht geeignet, da auch Anmeldeversuche von Angreifern korrekt sein könnten. Als Erkennungsmerkmal werden stattdessen Device Cookies auf *UAs* platziert, an denen Nutzer sich in der Vergangenheit bereits angemeldet hatten.
Wie in [Wal20] vorgeschlagen sind hierzu clientgestützte Bearer-Tokens einsetzbar, siehe Kapitel 2.2.4. Nach erfolgreicher Anmeldung wird ein Token ausgestellt, sofern der *UA* noch keinen besitzt. Er enthält folgende Claims (Felder):

- **uid**: Bindung des Tokens an den betreffenden Account
- **type**: Verwendungszweck zur Unterscheidung von Token-Typen, z. B. „rate-limit-device-cookie“
- **tid**: Nonce zur Token-Identifikation und temporären Token-Invalidierung

Abbildung 4.5 zeigt den Einsatz der Konstruktion: Sofern vorhanden schickt der Client den Token bei zukünftigen Anmeldeversuchen zum Account mit an den Server. Der Server wertet ihn aus Gründen der Ressourcenoptimierung nur aus, wenn der Account momentan einem Brute-Force-Angriff unterliegt. Falls dies der Fall ist, wird ein Anmeldeversuch nur mit gültigem Token gestattet. Jedoch ist auch die Anzahl an Fehlversuchen mit Token zu begrenzen. Deshalb ordnet der Server jedem vorgelegten Token via enthaltener *tid* einen Fehlerzähler zu. Überschreitet dieser einen festgelegten Schwellwert, wird der Token für eine Cool-Down-Zeit temporär invalidiert. Wie in [Wal20] ausgeführt, verhindert dies den Brute-Force durch Beschränkung des Request-Durchsatzes effektiv. Ein Angreifer kann durch gestohlene Device-Cookies den Durchsatz nur linear zur Anzahl der erbeuteten Tokens steigern, da sie dem Fehlerschwellwert unterliegen.

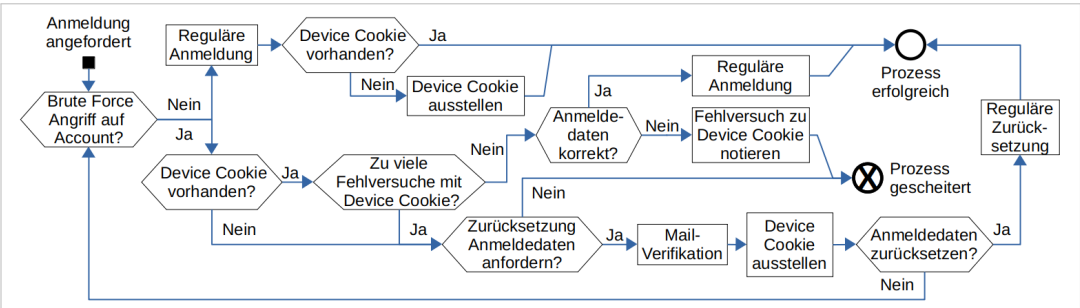


Abbildung 4.5: Einsatz von Device-Cookies als zusätzliche Authentifizierungsfaktoren zur kontrollierten Umgehung von Rate-Limiting-Beschränkungen
Quelle: Eigene Darstellung

Wie in Abbildung 4.5 dargestellt, ist auch im Falle eines Brute-Forces ohne Device-Token eine Anmeldung möglich. Dazu wird die Mail-Verifikation des Zurücksetzungs-Prozesses aus Kapitel 2.3.4 als Ersatzfaktor nachgenutzt, um einen Device-Cookie ausstellen zu können. Wie in Kapitel 4.1.4 herausgearbeitet, ist das Sicherheitsniveau hierbei identisch zur Passwortabfrage. Die Etablierung eines Rate-Limitings für die Mail-Verifikation wurde bereits in Kapitel 2.3.4 erläutert.

Einsatz zur Mail-Verifikation

Mehrere Protokollabläufe beinhalten eine Mail-Verifikation. So besteht u. a. die Registrierung aus zwei Prozessen: Registrierungsanfrage mit den angegebenen Stammdaten senden und Account via Link in der Verifikations-Mail bestätigen, siehe Kapitel 2.3.1. Zur Steigerung der Prozesssicherheit wird gefordert, dass sowohl Anforderung als auch Bestätigung einer Mail-Verifikation durch den gleichen *UA* erbracht werden. Um bei browserbasierten Anwendungen beide Teilprozesse an den gleichen *UA* zu binden, werden Device-Cookies eingesetzt. Anders als beim Rate-Limiting basieren diese nicht auf Bearer-Tokens:
Der Client erzeugt ein Opaque und speichert es im *UA* ab. Es wird der Verifikationsanforderung angehangen. Der Server nimmt das Opaque in die Konstruktion des Bearer-Tokens zur Mail-Verifikation auf. Der Nutzer ruft dann den Client via Verifikations-Link auf. Daraufhin prüft der Client zunächst die Übereinstimmung des Opaques in Bearer-Token und *UA*-Speicher. Nur bei Gleichheit erfolgt eine Übergabe des Verifikations-Tokens an den Server.
Dies verschlechtert potenziell die *UX*, da ein legitimer Verifikationsversuch mit einem anderen *UA* fehlschlägt. Falls der *UA* das Opaque vor Abschluss der Verifikation gelöscht hat, muss der Prozess erneut angestoßen werden. Allerdings wird ein Missbrauch der Mail-Verifikation verhindert: Ein Angreifer könnte z. B. einen Account mit der Mail-Adresse des Opfers anlegen. Aus Neugierde oder durch Social-Engineering könnte das Opfer den Verifikations-Link anklicken und würde damit sonst den Account für den Angreifer verifizieren.

Dieser Ansatz funktioniert, solange ein Angreifer die clientseitige Prüfung des Device-Cookies nicht umgehen kann. Dies ist jedoch u. a. in folgendem Szenario möglich: Der Nutzer gibt zur Registrierung seine privaten Stammdaten an. Bei der Mail-Adresse vertippt er sich, sodass ein Dritter die Verifikations-Mail erhält. Dieser kann als Angreifer nun die Verifikation vornehmen. Dazu entnimmt er das Opaque dem Verifikations-Token oder übergeht die clientseitige Prüfung. In dieser Position kann er die Anmeldedaten zurücksetzen und die privaten Stammdaten des Opfers einsehen. Dieses Szenario ist äußerst unwahrscheinlich. Es zeigt aber

anschaulich die Grenzen einer rein clientseitigen Validierung auf. Diese sind zu beachten, wenn ein Angreifer auf den Mail-Kanal Einfluss nehmen kann.

Deshalb wird die Prüfung des Device-Cookies unter Einsatz eines Challenge-Response-Verfahrens zum Server verlagert: Der Client wählt mittels *CSPRNG* einen geheimen, hochentropischen Zufallswert rnd und speichert ihn lokal im UA ab. Die Challenge leitet der Client als $rnd' := hash_{cli}^{fast}(rnd)$ ab. Wie in Kapitel 3.2.1 erläutert, genügt hierfür eine schnelle Hashfunktion, da rnd hochentropisch ist. Mit Beantragung der Verifikation wird rnd' an den Server gesendet und von diesem in den auszustellenden Bearer-Token aufgenommen. Zur Bestätigung des Verifikationsprozesses ist rnd als Response zusammen mit dem Token am Server vorzulegen. Der Server prüft abschließend durch Hashing von rnd, ob der Wert rnd' entsteht. Da rnd' im Verifikations-Token enthalten ist, kann ein Angreifer keine abweichende Challenge präsentieren. Abbildung 4.6 fasst den Prozess am Beispiel eines Registrierungsvorgangs zusammen.

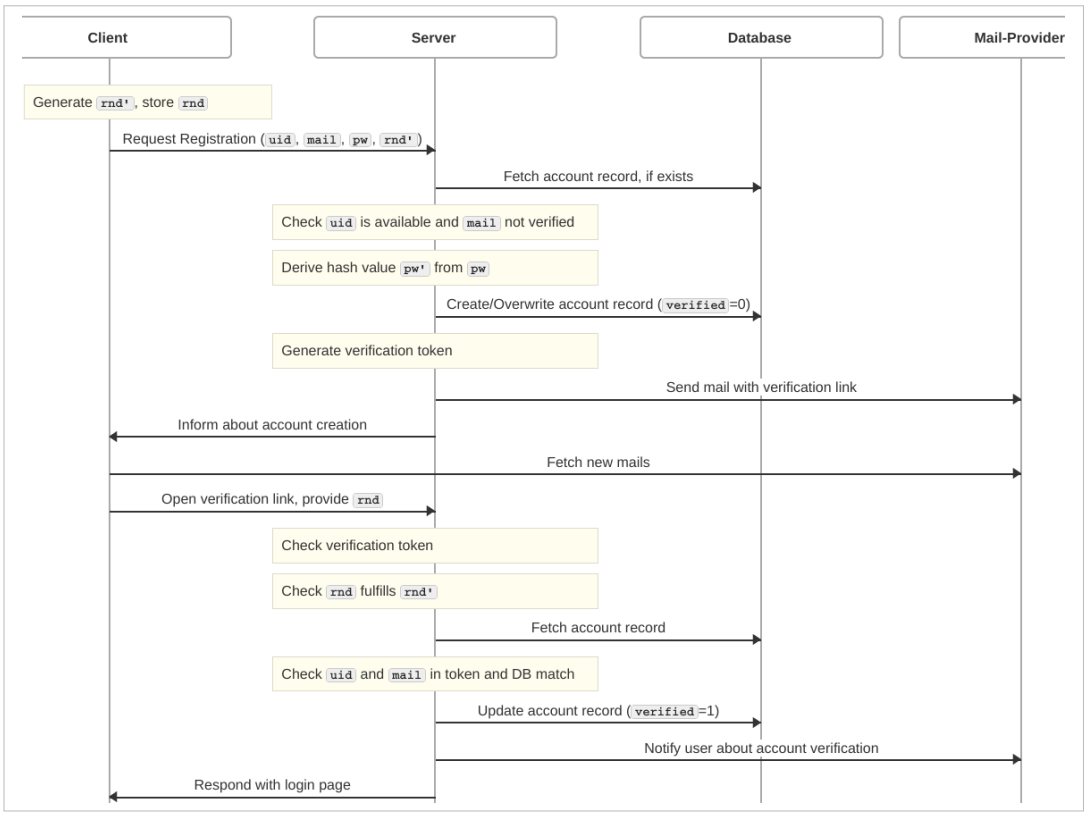


Abbildung 4.6: Einsatz von Device-Cookies zur Mail-Verifikation mit Cookie-Prüfung durch den Server am Beispiel des Registrierungsprozesses
Quelle: Eigene Darstellung

Einsatz zur iterativen MFA

Wie die Mail-Verifikation erfordert die iterative *MFA* mehrere Kommunikationsschritte zwischen Client und Server. Es gilt sicherzustellen, dass beide Kommunikationsteilnehmer in jedem Prozessschritt die gleichen wie im vorherigen sind. Hierzu wäre das *TLS* Channel Binding, wie in Kapitel 3.3.3 zu SCRAM-PLUS vorgestellt, nutzbar. Allerdings würde eine Layer-Kopplung benötigt, die möglicherweise nicht alle Systeme unterstützen. Um die Deployability nicht einzuschränken, werden Device-Cookies eingesetzt. Sie erlauben eine Realisierung durch die Applikation ohne Layer-Abhängigkeiten zu schaffen.

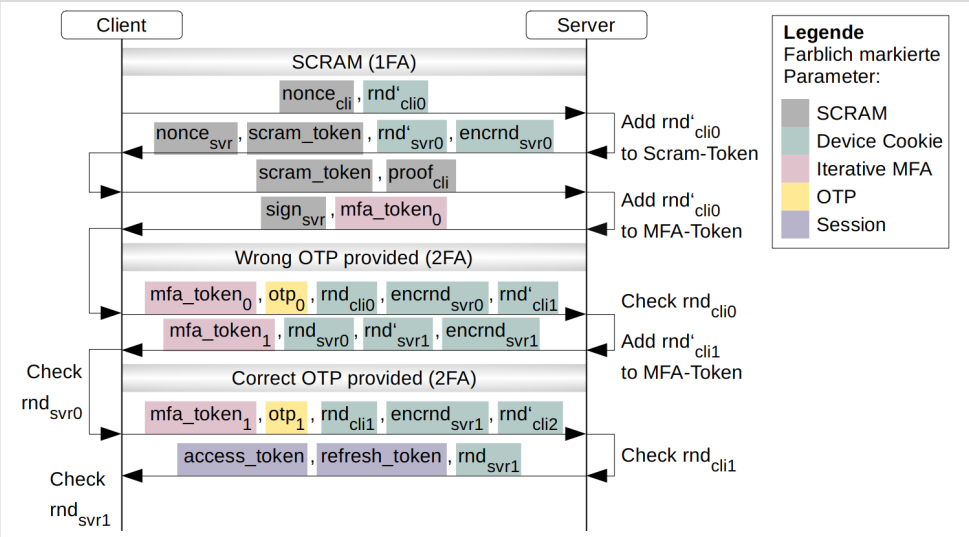


Abbildung 4.7: Einsatz von Device-Cookies zur Absicherung der iterativen MFA am Beispiel eines 2FA-Login-Vorgangs mit SCRAM und OTP
Quelle: Eigene Darstellung

In jeder *MFA*-Iteration $i \in \mathbb{N}_0$ erzeugen Client und Server Schlüsselpaare: $(rnd_{cli_i}, rnd'_{cli_i})$ und $(rnd_{svr_i}, rnd'_{svr_i})$.
Dazu nutzen sie jeweils einen *CSPRNG* und $hash^{fast}$ nach dem oben erläuterten Verfahren. Zu Beginn einer Iteration werden rnd'_{svr_i} und rnd'_{cli_i} ausgetauscht und mit der nächsten dann rnd_{svr_i} und rnd_{cli_i} sowie ggf. $rnd'_{svr_{i+1}}$ und $rnd'_{cli_{i+1}}$.
Ein *Eavesdropper*, der selbst einen Faktor kennt, wird gehindert die Authentifizierung abzuschließen. Allerdings müsste der Server zwischen Requests den Zustand rnd_{svr_i} speichern. Um dies zu vermeiden, wird der Wert verschlüsselt an den Client ausgelagert: $encrnd_{svr_i} := enc_{sym}(rnd_{svr_i}, secret_{svr})$
Dies ist möglich, da der Client keinen Vorteil davon hätte, dem Server eine andere verschlüsselte Response zu präsentieren. Da die Kette mit zufällig gewähltem, annähernd einmaligem Startwert fortlaufend ist, sind gesammelte Challenge-Response-Paare des Servers auch nicht für andere *MFA*-Vorgänge nutzbar. Damit muss der Server nur das statische $secret_{svr}$ geheimhalten. Solange dies der Fall

ist, kann er nicht im Laufe einer iterativen *MFA* impersoniert werden. Abbildung 4.7 zeigt das Vorgehen exemplarisch für einen *2FA*-Login mit *SCRAM* und *OTP*, wobei der *OTP* beim ersten Versuch falsch eingegeben wird. Allerdings ist das durch den Server-Check erzielbare Sicherheitsniveau geringer als das des Client-Checks: Da der Identitätsnachweis via Device-Cookie durch einen Geheimnisnachweis mittels Bearer-Token ersetzt wurde, genügt die einmalige Kenntnis des statischen $secret_{svr}$ zur Impersonierung. Alternativ zu diesem symmetrischen Ansatz bietet die Verwendung asymmetrischer Schlüsselpaare den Vorteil, dass sie nur einmalig für den Prozess zu wählen sind, siehe Kapitel 4.3.1.

4.2.4 Risiko-Bewertung

Zu clientseitigen Passwort-Generatoren wurde in Kapitel 4.2.1 herausgearbeitet, dass ihr sicherer Einsatz in Form eines Features von Passwort-Managern besteht. Die Sicherheit eines Passwort-Managers hängt neben einem starken Main-Passwort maßgeblich von der Verwendung durch den Nutzer ab, weshalb dieser Aspekt keiner allgemeinen Risiko-Einschätzung zugänglich ist. Jedoch eignet sich das Verfahren durchaus zum Aufbau einer starken Client-*2FA*, siehe Kapitel 4.2.1.

Tabelle 4.1: Risiko zum Phishing für Passwörter in Webapplikationen

D	5/5	Vollständige, permanente Accountübernahme
R	5/5	Betrieb von Phishing-Websites und -Kampagnen
E	4/5	Nutzer identifizieren, Typosquatting betreiben
A	4/5	Betrifft alle Opfer einer Phishing Kampagne
$\sum$	**18/20**	$\Rightarrow$ **Hohes Risiko**

Tabelle 4.2: Risiko zum Phishing für OTPs in Webapplikationen

D	5/5	Vollständige, einmalige Accountübernahme
R	5/5	Betrieb von Phishing-Websites mit mehrfacher *OTP*-Eingabe
E	4/5	Nutzer identifizieren, Typosquatting betreiben
A	4/5	Betrifft alle Opfer einer Phishing Kampagne
$\sum$	**18/20**	$\Rightarrow$ **Hohes Risiko**

Kapitel 4.2.2 legt das Phishing-Problem von Passwörtern und *OTPs* aufgrund seiner Brisanz ausführlich dar. Bei einer Webapplikation erlaubt eine Passwort-*1FA* dem Angreifer durch erfolgreiches Phishing die Accountübernahme, siehe Tabelle 4.1. Zusätzlich kann er auch innerhalb der Applikation geschützte Aktionen ausführen, die separat über das Passwort zu authentifizieren sind, siehe Kapitel 2.3.5. Eine *2FA* mit Passwort und *OTP* könnte zumindest letzteres verhindern, siehe Tabelle 4.2. Allerdings darf der Nutzer dann nicht mehrere *OTPs* auf der Phishing-Webseite eingeben. Dies kann der Angreifer jedoch trivial provozieren, indem er dem Nutzer anzeigt, dass eine korrekte *OTP*-Eingabe falsch

war und der Nutzer daraufhin den nächsten *OTP* eingibt. Daher sind beide Varianten (Tabelle 4.1 und 4.2) beim Einsatz für Webapplikationen mit einem hohen Risiko verbunden. Eine *2FA* mit *OTPs* ist für Webapplikationen nicht wesentlich sicherer als eine *1FA* mit Passwort.

Tabelle 4.3: Risiko zum Belauschen von OTP-Eingaben

D	1/5	Kenntnis eines nicht mehr verwendbaren OTPs
R	4/5	Abfangen bei Authentifizierungen mit OTP möglich
E	2/5	Einbruch in TLS, Nutzer ausspionieren, Endgerät infizieren
A	2/5	Angriff gegen einen Nutzer, parallelisierbar
$\sum$	**9/20**	⇒ **Niedriges Risiko**

Tabelle 4.4: Risiko für TOTP zum Replay alter OTP-Werte

D	4/5	Aushebelung von MFA-Faktoren
R	1/5	Manipulation der Systemzeit des Servers
E	1/5	Sammlung übertragener OTPs als Eavesdropper
A	1/5	OTP-Nutzer mit kompromittierten Verbindungen
$\sum$	**7/20**	⇒ **Niedriges Risiko**

Auf andere Plattformen als das Web, z. B. Desktop, ist die Betrachtung aus Tabelle 4.2 nicht übertragbar. Hier bieten *OTPs* einen wirksamen Schutz gegen Angreifer, die Nutzereingaben oder den Transportkanal belauschen, siehe Tabelle 4.3. Voraussetzung bleibt, dass die originale Übertragung schneller den Server erreicht, als eine durch den lauschenden Angreifer initiierte. Im Gegensatz dazu würde in diesem Szenario für Passwörter die Betrachtung aus Tabelle 3.3 in Kapitel 3.1.3 gelten, welche ein mittleres anstatt niedriges Risiko zeigt.
Bei *TOTP* kommt durch Einbeziehung der Zeit ein weiterer Angriffsvektor hinzu: Durch Manipulation der Serverzeit könnte ein Angreifer alte *TOTP*-Werte erneut einreichen und damit die Nonce-Eigenschaft verletzen. Dies ist bei einer geschützten Server-Infrastruktur jedoch nur schwer ausnutzbar, siehe Tabelle 4.4.

Von einem *DB*-Leak sind auch die *OTP*-Seeds betroffen. Da diese äquivalent zum Peppering der Passwort-Hashes durch Verschlüsselung geschützt sind, wird die Betrachtung bezüglich der Pepper-Kenntnis des Angreifers unterschieden, siehe Tabelle 4.5 und 4.6. Für einen verschlüsselten *OTP*-Seed lassen sich mögliche Klartexte nur bestimmen, sofern zugehörige *OTP*-Werte zahlreich bekannt sind. Das Risiko bei Pepper-Kenntnis des Angreifer ist für *OTPs* und Passwörter ähnlich, siehe hierzu Tabelle 3.2 in Kapitel 3.1.3. Dies ist dem Umstand geschuldet, dass *OTPs* in diesem Szenario allein keinen Schutz bieten. Da der *OTPs*-Einsatz jedoch nur in Kombination mit weiteren Faktoren wie Passwörtern erfolgt, sind deren Schutzmaßnahmen zur Beurteilung der Ausnutzbarkeit heranzuziehen, siehe Tabelle 3.2. Wie in Kapitel 3.3.4 ausgeführt, ist die Einschätzung in Tabel-

le 4.6 jedoch vor dem Hintergrund der privilegierten Angreifer-Möglichkeiten des Szenarios zu bewerten.

Tabelle 4.5: Risiko zu OTPs für DB-Leak ohne Pepper

D	3/5	Eingrenzung möglicher OTP-Seeds
R	1/5	Als MITM übertragene OTP-Werte abfangen und auswerten
E	2/5	Zugang zu DB-Auszug notwendig
A	2/5	Angreifer muss je Nutzer OTPs abfangen, parallelisierbar
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Tabelle 4.6: Risiko zu OTPs für DB-Leak mit Pepper

D	4/5	Aushebelung von MFA-Faktoren
R	2/5	Angreifer muss weitere MFA-Faktoren ermitteln
E	1/5	Zugang zur Server-Infrastruktur
A	5/5	Betrifft sämtliche OTP-Seeds in der DB
$\sum$	**12/20**	$\Rightarrow$ **Mittleres Risiko**

Durch die Einführung der Device-Cookies, siehe Kapitel 4.2.3, ergeben sich keine Threats, sondern nur Security-Verbesserungen. Ihr Einsatz ist aus Perspektive der Protokollsicherheit uneingeschränkt zu empfehlen.

4.3 Asymmetrische Verfahren

Asymmetrische Verfahren verwenden als Primitiv selbstredend asymmetrische Kryptografie. Grundlage bilden Schlüsselpaare der Form (key_{pub}, key_{pri}) mit key_{pub} als Public-Key und key_{pri} als Private-Key. Der private Schlüssel sollte optimalerweise nie den Erzeugungskontext verlassen, ist also z.B. nicht im Netzwerk zu übertragen. Der öffentliche Schlüssel wird an Kommunikationspartner verteilt, die den Besitz des privaten Schlüssels verifizieren müssen. Für diesen Prozess gilt es, einen sicheren Übertragungskanal zu wählen und die Authentizität des Absenders zu überprüfen. Andernfalls besteht die Gefahr, einen vom Angreifer erzeugten Schlüssel zu akzeptieren. Nachfolgend werden Nutzungsmöglichkeiten zur *MFA* vorgestellt.

4.3.1 Asymmetrische Device-Cookies

In Kapitel 4.2.3 wurden Device-Cookies als Technik zur User-Agent-Bindung vorgestellt. Dieser Ansatz wird nachfolgend um die Möglichkeiten asymmetrischer Verfahren erweitert.

Realisierung durch den Transport-Layer

Mit Aufbau einer *TLS*-Verbindung zum Server generiert der Client ein asymmetrisches Schlüsselpaar und propagiert den Public-Key über den gesicherten *TLS*-Kanal zum Server. Im Kontext der Verbindung hält der Client den Private-Key und der Server den Public-Key. Die Applikation kann diese Struktur zur Nachrichten-Verifikation nutzen: Der Client signiert seine Nachrichten mittels Private-Key und der Server prüft die Signaturen mittels Public-Key. Dadurch werden Replay-Attacken in anderen Verbindungen verhindert, solange der Angreifer nicht den Private-Key extrahieren kann. Dieser Ansatz wird in [PNB+18b] als Token Binding spezifiziert und in [PNB+18a] um eine Einbindung in *HTTP* erweitert.

Token Binding als Authentifizierungsschema kann auch wechselseitig eingesetzt werden. In Kapitel 4.2.3 wurde anhand der iterativen *MFA* eine alternative Möglichkeit erläutert, dies rein durch die Applikation zu bewerkstelligen. Dabei waren für jeden Nachrichtenaustausch neue Schlüssel zu wählen, was beim Token Binding entfällt. Nachteilig war außerdem die starke Kopplung zwischen Kommunikation der Applikation und Nachrichten zur Signierung und Verifikation der Teilnehmer. Token Binding bietet hier eine architektonisch besser entkoppelte Alternative und ist deshalb zu bevorzugen. Allerdings ist das Token Binding eine *TLS*-Erweiterung, die nicht von allen Teilnehmern unterstützt werden muss. *UAs* ohne Support würden nicht profitieren.
Das Token Binding kann als Weiterentwicklung des in Kapitel 3.3.3 vorgestellten Channel Bindings angesehen werden: Channel Binding exponiert lediglich Schlüsselmaterial an die Applikation, damit diese ein Verifikationssystem darauf aufbauen kann. Im Gegensatz dazu schafft Token Binding bereits die geeignete Grundlage, eine Verifikation in die Applikation zu integrieren. Allerdings ist der Overhead einer *TLS*-Verbindung durch das Einfügen von Schlüsselmaterial größer als bei Nachnutzung des vorhandenen Fingerprints. Bei Geräten mit geringer Speicheranbindung, z. B. *Internet of Things (IoT)*, können parallele Verbindungen damit zu Speichermangel führen. Für eine abhängigkeitslose Technologiewahl ist Token Binding gegenüber Channel Binding zu präferieren.

Realisierung durch die Applikation

Anstatt das asymmetrische Schlüsselpaar an die kurzlebige Verbindung zu binden, lässt es sich auch als langlebiger *MFA*-Faktor zur Applikation anwenden. Zur Faktor-Registrierung entsprechend Kapitel 4.1.4 generiert der Client ein Schlüsselpaar (key_{pub}, key_{pri}), speichert key_{pri} lokal und sendet key_{pub} an den Server. Im Backend wird eine Faktor-ID fid generiert. In der *DB* wird ein Record bestehend

aus (uid, fid, key_{pub}) persistiert. Der Client speichert den Record (uid, fid, key_{pri}) im *UA*. Die Aufnahme der *uid* im *UA*-Record dient der Unterscheidung von Private-Keys verschiedener Accounts.

Das Verfahren ermöglicht dem Nutzer sein Endgerät als „vertrauenswürdig“ zu markieren. Eine *2FA* kann beispielsweise wie folgt gestaltet werden: Auf unbekannten Geräten werden Passwort und *OTP* abgefragt. Markiert der Nutzer das Gerät als vertrauenswürdig, so entfällt zur Authentifizierung die *OTP*-Angabe, da stattdessen der Private-Key ausgewertet wird. Einem Endgerät wird die Vertrauenswürdigkeit entzogen, indem der zum Faktor zugehörige Record aus der *DB* gelöscht wird. Registrierung und Löschung dieser Faktoren erfolgen im Kontext einer gültigen Session nach dem Protokoll aus Kapitel 4.1.4.

Zur Implementierung des Verfahrens wird die iterative *MFA* aus Kapitel 4.1.2 um Challenges erweitert, die mittels *CSPRNG* zu erzeugen sind. In die Konstruktion eines *MFA*-Tokens durch den Server wird jeweils eine Challenge aufgenommen. Zur Authentifizierung signiert der Client mittels Private-Key die Challenge und sendet seine Response zusammen mit der fid an den Server zurück. Der Server kann nun prüfen, ob die fid zum Nutzer hinterlegt und die Signatur gültig ist. Im Verlauf der iterativen *MFA* darf nur einmalig ein Private-Key aus dem *UA*-Speicher präsentiert werden, da nur ein Schlüssel je Endgerät und Account hinterlegt wird. Ansonsten ist von einem Angriff mit gestohlenen Keys auszugehen. Eine Mehrfach-Präsentation wird durch Ergänzung des *MFA*-Tokens um ein entsprechendes Flag verhindert.

Gegenüber symmetrischen bietet dieses Verfahren strukturelle Vorteile: Anders als bei Passwörtern ist aufgrund der Schlüssel-Auftrennung kein aufwändiges Hashing notwendig. Im Gegensatz zu *OTPs* muss das Geheimnis serverseitig nicht im Klartext vorhanden sein. Ein Überbau wie *SCRAM* entfällt, da das Merkmal direkt zur Durchführung eines Challenge-Response-Verfahrens geeignet ist, vergleiche Kapitel 3.3.3.
Eine symmetrische Verschlüsselung der Private-Keys im *UA* zum Schutz vor Diebstahl ist uneingeschränkt sinnvoll. Hierzu kann das Account-Passwort nachgenutzt werden. Von einer direkten Anwendung des Passwortes ist allerdings abzusehen: Ein Angreifer im Besitz der *DB*-Inhalte und des Private-Keys aus dem *UA* könnte eine Dictionary-Attack zur Ermittlung des Passwortes über die symmetrische Verschlüsselung starten. Stattdessen ist die Verschlüsselung mittels Passwort-Hashwert aus einer langsamen Hashfunktion zu präferieren. Ein solcher Hashwert existiert bei Anwendung von Client-Hashing bereits im Client, siehe Kapitel 3.
Die Verschlüsselung erzeugt durch Kopplung an das Passwort ein *UX*-Problem:

Im Falle eines Passwortwechsels wird die weitere Verwendung der Schlüssel auf anderen Geräten verhindert. Dies kann jedoch auch gewollt sein, falls der Passwortwechsel als Reaktion auf einen vermuteten Angriff erfolgt.
Eine symmetrische Verschlüsselung der Public-Keys in der *DB* ist nur in Sonderfällen sinnvoll: Dem Merkmal Passwortkenntnis ist inhärent, dass Nutzer die Kenntnis abstreiten können. Für Public-Key-Verfahren ist die Abstreitbarkeit erzielbar, indem der Schlüssel-Zusammenhang nur schwer herzustellen ist. Ein Dritter könnte die Public-Keys aus einem Data-Breach verschlüsselt kennen. Ohne Zugriff auf das Live-System kann er nicht nachweisen, dass ein Private-Key von einem betreffenden Nutzer-Gerät zu einem bestimmten Account gehört.

Das vorgestellte Konzept wurde als Ersatz zur passwortbasierten Web-Authentifizierung in [FHT15] als *HTTP Origin-Bound Authentication (HOBA)* vorgeschlagen. Die namensgebende Origin-Bindung wird durch den *UA* erzielt: Webbrowser stellen für jede Origin einen isolierten Persistenzbereich bereit, der für andere Origins unzugänglich ist. Durch clientseitige Ablage der Private-Keys im *UA*-Speicher wird diese Bindung hergestellt. Damit ist das Verfahren im Gegensatz zu *OTPs* und regulären Passwörtern nicht vom Problem des *Origin-Spoofings* betroffen, siehe Kapitel 4.2.2.
Vom Schutz durch Origin-Bindung profitiert die clientseitige Ablage der Bearer-Tokens ebenfalls, siehe Kapitel 2.2.4. Im Gegensatz zu den symmetrischen Bearer-Tokens ist *HOBA* als *CRAM* zudem geeignet, Replay-Attacken zu verhindern.

Durch die *UA*-Kopplung ist *HOBA* in die Mechanismen und Abläufe des *UA* integriert. So kann ein Nutzer z. B. im Webbrowser die Website-Daten löschen, um Private-Keys zu entfernen, oder auch den sogenannten privaten Modus als sterile Browser-Umgebung nutzen. Eine *UA*-Integration ist alternativ auch über Client-Zertifikate möglich, wie nachfolgend vorgestellt.

4.3.2 Zertifikat-Authentifizierung durch den User-Agent

In Kapitel 2.2.1 wurde ein zertifikatsbasiertes *PKI*-System mit *TLS* vorgestellt und *mTLS* als Verfahren zur wechselseitigen Authentifizierung mittels *TLS* benannt. *mTLS* erlaubt die Realisierung des zuvor erläuterten Authentifizierungsverfahrens mit Zertifikaten anstatt bloßen asymmetrischen Schlüsseln. Hierzu wird eine Root-*CA* aufgebaut, die zur Ausstellung von Nutzer-Zertifikaten dient. Ein Nutzer-Zertifikat kann je Account oder je Account und Endgerät ausgestellt werden. Letzteres bietet eine feingranulare Steuerung bei höherem Organisationsaufwand. Zur Verifikation von Nutzer-Zertifikaten wird im Backend dann nur das Root-Zertifikat anstatt der einzelnen Public-Keys benötigt. Die Validitäts-

prüfung eines Nutzer-Zertifikats erfolgt via *CRL* bzw. *OCSP*, siehe Kapitel 2.2.1. Außerdem findet durch Nutzung der Zertifikatsstruktur, im Gegensatz zum Ansatz aus dem vorherigen Abschnitt, eine automatische Invalidierung nach Ablauf des Gültigkeitszeitraums eines Zertifikats statt. Damit wird dem Gefahrenpotenzial von Private-Keys begegnet, die verloren oder Dritten bekannt geworden sind. Es gilt bereits beim Aufbau des Systems einen Prozess zur Zertifikatserneuerung zu etablieren, um Nutzer nicht nach Ablauf ihrer Zertifikate auszusperren.

Auf *mTLS* ist wie erläutert die Sicherheitsbetrachtung des vorherigen Abschnitts anwendbar, welche die hohe Sicherheit des Verfahrens zeigt. Die Deployability hängt von der Unterstützung durch den *TLS*-Kanal zwischen *UA* und Backend ab. Da das Verfahren bereits 2008 in [RD08, S. 55 ff.] für *TLS* 1.2 spezifiziert wurde, ist eine hohe Verbreitung anzunehmen. Die Technik *mTLS* scheint damit zur Nutzung von Zertifikaten als *MFA*-Faktor der Kategorie Besitz dienlich.

Allerdings passt das Nutzungsmodell nicht zum angedachten Use-Case: Insbesondere bei Webbrowsern wird *mTLS* vorrangig zur Bildung einer Vertrauensstellung zwischen Client-Geräten und Backend genutzt. Ein echtes Identitätsmanagement mit Funktionen für Login, Logout oder auch Accountwechsel ist nicht abbildbar. Stattdessen ist das Nutzungsmodell eher auf folgende Szenarien ausgelegt:

- Aufbau eines browserbasierten *Virtual Private Network (VPN)*: Der Zugang zum geschützten Netzwerk ist via *mTLS* abgesichert. Nur mit Zugang zum geschützten Netzwerk können Nutzer eine Authentifizierung gegen die Applikation in diesem abgeschotteten Netz durchführen.
- Starke Absicherung von *M2M*-Kommunikation über angreifbare Kanäle.

Diese beiden Realisierungen entsprechen auch der Backend-Architektur: Wie in Kapitel 3.3.3 bereits thematisiert, sollte der Reverse-Proxy aus Authentifizierungsvorgängen der dahinterliegenden Applikationen herausgehalten werden, um die architektonische Kopplung möglichst lose zu halten. Die Verwendung von *mTLS* als *MFA*-Faktor würde eine starke Abhängigkeit der Applikation von einem spezialisierten Reverse-Proxy schaffen, da *mTLS* im *TLS*-Kanal zwischen Client und Server abläuft. Damit wäre die Trennung der Zuständigkeiten (engl. Separation of concerns) im Backend verletzt.
Vom Einsatz von *mTLS* zur Applikations-Authentifizierung ist aufgrund der Probleme in Frontend und Backend abzusehen. Nachfolgend werden geeignetere Integrationsmöglichkeiten für *MFA*-Faktoren via Public-Key-Verfahren vorgestellt.

4.3.3 Security-Tokens

Ein Security-Token ist eine Hardwarekomponente, auf der kryptografisches Schlüsselmaterial gespeichert und verarbeitet werden kann. Als Zweitgerät kommuniziert es mit dem Erstgerät des Nutzers über einen Hardware-Bus oder via Funk. Durch diese Architektur wird eine Gerätetrennung erzielt. Diese muss jedoch nicht physikalisch ausgeprägt sein:
Beispielsweise verfügen moderne Personal Computer und Smartphones in der Regel über einen integrierten *TPM*-Chip, der als Security-Token nutzbar ist. Damit kann das Konzept der asymmetrischen Device-Tokens aus Kapitel 4.3.1 weiter abgesichert werden: Der Private-Key wird nicht mehr im *UA* gespeichert, wo er potenziell von einem Programm auf dem Host auslesbar ist. Stattdessen wird er durch den *TPM*-Chip erzeugt und dort gespeichert. Da die Speicherkapazitäten eines *TPMs* begrenzt sind, kann alternativ eine Festplattenspeicherung bei symmetrischer Verschlüsselung durch einen Key im *TPM* erfolgen.
Durch die Gerätetrennung ist es selbst für das Betriebssystem nicht mehr möglich, den Private-Key auszulesen. Ein Zugriff auf via Security-Token abgesicherte Accounts bedingt den Zugang zum Security-Token, da das Schlüsselmaterial nicht mehr kopierbar ist. Security-Token werden optimalerweise so konstruiert, dass auch eine Extraktion von Schlüsselmaterial nicht zerstörungsfrei möglich ist.

Gerätegebundene Security-Tokens wie *TPMs* sind geeignet, jeweils ein Endgerät abzusichern. Wie in Kapitel 4.3.1 beschrieben, markiert der Nutzer das Gerät durch Ablage eines Keys als vertrauenswürdig. Währenddessen sind ausschließlich an Nutzer gebundene Security-Tokens, wie Chip-Karten, aufgrund ihrer Portabilität geräteübergreifend einsetzbar und erzielen damit eine bessere Usability.

Ein nutzergebundenes Zweitgerät könnte z. B. statischen Passwortbesitz oder *OTP*-Seeds verdeckt speichern. Zum Einfügen von Passwort bzw. *OTP* könnte es sich z. B. als *Universal Serial Bus (USB)*-Tastatur am Erstgerät anmelden. Die Usability wäre gegenüber manueller Eingabe verbessert und die Deployability geringfügig verschlechtert, da nicht jedes Endgerät über *USB* o. ä. Schnittstellen verfügt. Allerdings gilt auch weiterhin die Sicherheitsbetrachtung zu Passwortbesitz und *OTPs* gleichermaßen, siehe Kapitel 4.2.1 und 4.2.2. Insbesondere die Anfälligkeit für *Origin-Spoofing* bleibt erhalten. Deshalb sollte das Zweitgerät auch origin-gebundene Public-Key-Verfahren unterstützen. Hierzu wird nachfolgend exemplarisch der mit Fokus auf Webapplikationen entworfene Standard FIDO2 diskutiert. Als Alternative ist an dieser Stelle auch *Personal Identity Verification (PIV)* zu nennen, welches Smartcards nutzt, wie z. B. Chip-Karten. Es wird allgemein für physische und virtuelle Zugangskontrollsysteme eingesetzt.

FIDO2

FIDO2 ist der aktuelle Standard der Allianz *Fast Identity Online (FIDO)* zur Web-Authentifizierung mittels Security-Tokens. Er beerbt den früheren Standard *Universal 2nd Factor (U2F)*. Wie in Abbildung 4.8 dargestellt, besteht FIDO2 aus den Protokollen *Client to Authenticator Protocol (CTAP)* und *Web Authentication API (WebAuthn)*.

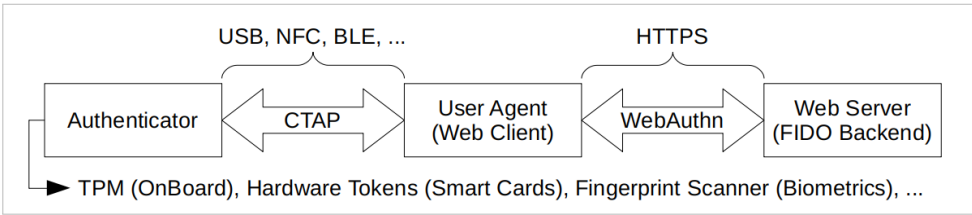


Abbildung 4.8: Architektur und Protokolle des FIDO2-Standards
Quelle: Eigene Darstellung

CTAP regelt die Kommunikation mit dem Zweitgerät des Nutzers, nachfolgend Authenticator genannt. Dieses kann, wie oben erläutert, sowohl an den Nutzer als auch das Endgerät gebunden sein. Als Transportprotokoll für nutzergebundene, externe *CTAP*-Geräte sind exemplarisch *USB*, *Near Field Communication (NFC)* sowie *Bluetooth Low Energy (BLE)* zu nennen.[BHJ+21]
WebAuthn spezifiziert eine *API* für den Datenaustausch zwischen *UA* und dem Webclient der Applikation. Dies ist notwendig, da eine Webseite zur Umsetzung des *PoLP* keine Berechtigung zum direkten Hardware-Zugriff besitzt. Die Webseite kann lediglich über die gebotenen Schnittstellen der *UA-Sandbox* mit dem Authenticator interagieren. Hieraus ergibt sich direkt das Deployability-Problem von FIDO2: Der *UA* muss nicht nur *WebAuthn* unterstützen, sondern auch unterschiedliche Geräteklassen via *CTAP* ansprechen können. Die Güte von FIDO2-Authentifizierungen ist damit nicht nur anhand der Einbindung in die Webapplikation zu beurteilen. Sie hängt ebenfalls maßgeblich von der Güte der *UA*-Implementierung sämtlicher betrachteter *UAs* sowie *UA*-Versionen ab. Der diesbezügliche Test in Kapitel 5.2.4 zeigt jedoch bereits eine gute Unterstützung durch die getesteten Systeme.

Nachfolgend werden die Protokollschritte von FIDO2 erläutert, wobei zur Übersichtlichkeit das generelle Funktionsprinzip zugunsten der tatsächlichen Parameterausgestaltung aus der Spezifikation [HJL+21] dargestellt wird.

Registrierung Die Registrierung eines Authenticators zum Account wird über das Protokoll aus Kapitel 4.1.4 in die Applikation integriert, respektive im Kontext einer gültigen Sitzung erbracht. Der Funktionsablauf ist ähnlich zu dem in

Kapitel 4.3.1 erläuterten Protokoll *HOBA* mit einigen Erweiterungen, wie der Parametrisierung von *CTAP*. Wie in Abbildung 4.9 dargestellt, erzeugt der Server für das *CRAM* zunächst eine Challenge. Da der Webbrowser den am Authenticator vorgelegten Domainnamen kontrolliert, wird eine Origin-Bindung des auszustellenden Credential erzielt. Eine Umgehung bedarf eines Bugs im oder der Kompromittierung des *UA*. Am Authenticator muss der Nutzer bestätigen, dass ein Credential für die Webseite ausgestellt werden soll. Folgende, durch die Applikation vorzugebende Abstufungen sind möglich:

- ***User Presence (UP)***: Bestätigung der Anwesenheit eines Menschen, z. B. durch mechanischen Druck auf einen Knopf am Authenticator.
- ***User Verification (UV)***: Authentifizierung gegenüber dem Authenticator gefordert, z. B. via Code-Eingabe oder biometrischem Fingerabdruck.

Durch *UP* bzw. *UV* wird sichergestellt, dass ein Mensch der Aktion zustimmt. Ansonsten könnte eine bösartige Software auf dem Client-System unbemerkt beliebige Challenges signieren lassen. Ein *TPM* bietet beispielsweise weder *UP* noch *UV* und sollte deshalb nur in Kombination mit einem durch den Nutzer erbrachten Merkmal als Authenticator eingesetzt werden.

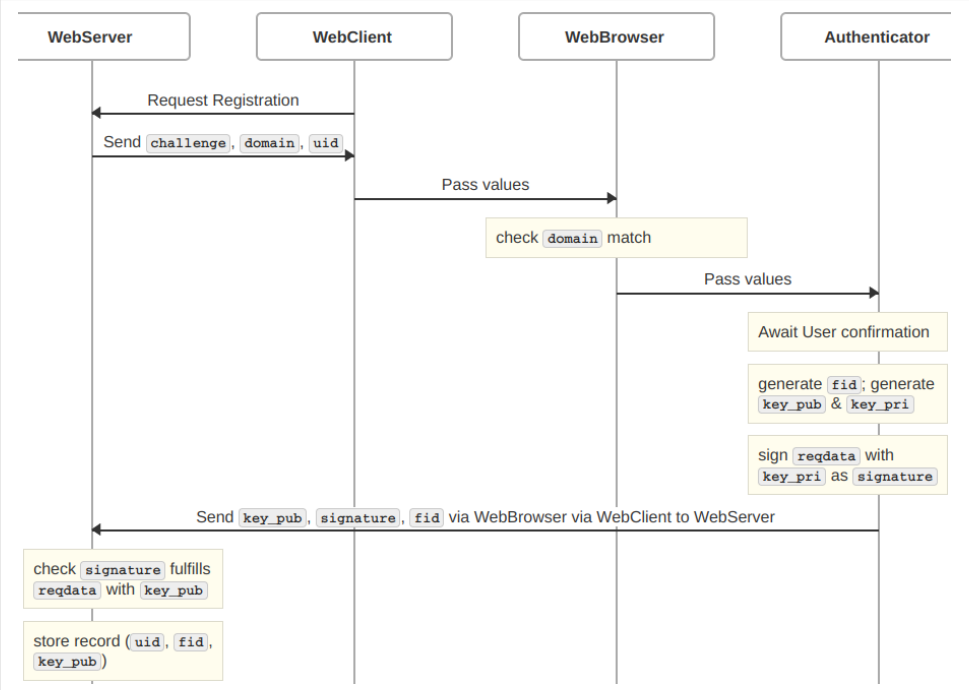


Abbildung 4.9: Protokoll zur Faktor-Registrierung via FIDO2
Quelle: Eigene Darstellung

Der Authenticator erzeugt eine zufällige fid (Faktor-ID) sowie ein Schlüsselpaar (key_{pub}, key_{pri}) für das neue Credential. Der key_{pri} ist durch den Authenticator in Abhängigkeit von *domain*, *uid* und fid ermittelbar, welche in Abbildung 4.9

als *reqdata* zusammengefasst sind. Als weitere Parameter fließen in *reqdata* Angaben zu eingesetzten kryptographischen Verfahren sowie Flags ein, u. a. ob *UP* oder *UV* angefordert wurde. Hinzu kommt eine optionale Nonce in Form eines Counters, der nach jeder Signierung inkrementiert wird.
Durch diese Struktur sind zu einem Credential keine Informationen auf dem Authenticator zu speichern, sondern nur ein einzelner geheimer Schlüssel. Aufgrund dieser Zustandslosigkeit ist die Anzahl erzeugbarer Credentials nicht durch persistente Speicherreserven des Authenticators begrenzt. Abschließend signiert der Authenticator die Challenge via key_{pri} und sendet die Antwort an den Server. Dieser kann nun anhand des übermittelten key_{pub} prüfen, ob die Signatur zu seiner Challenge sowie Domainname und fid korrekt sind. Im Erfolgsfall legt er als Authentifizierungsfaktor das Tupel (uid, fid, key_{pub}) in der *DB* ab.

Authentifizierung Die Authentifizierung verläuft ähnlich zur Registrierung: Der Server sendet neben der Challenge auch die Liste zum Account ausgestellter FIDO2-Credentials an den Client, welcher sie zusammen mit *uid* und *domain* an den Authenticator weitergibt. Während die Nutzerbestätigung (*UP* bzw. *UV*) eingeholt wird, kann der Authenticator, falls technisch möglich, die *uid* anzeigen, sodass der Nutzer in der Lage ist, sie zu verifizieren. Nach Nutzerbestätigung kann der Authenticator aus den bereitgestellten Angaben ermitteln, ob ihm einer der präsentierten Public-Keys bekannt ist, respektive ob er über ihn verfügt. Wenn dies der Fall ist, signiert er die Challenge mit dem passenden Private-Key. Die Signatur sowie genutzte fid werden an den Server übermittelt, welcher seinerseits die Signaturprüfung mit dem zur fid in der *DB* hinterlegten key_{pub} vornimmt.
Dieses Authentifizierungsschema wird in die iterative *MFA* aus Kapitel 4.1.2 als nicht-primärer Faktor eingebunden. Als primärer Faktor eignet sich das Protokoll nicht, da vom Server die Public-Keys bezogen werden müssen und sonst durch die Informationspreisgabe eine Security-Schwachstelle vorliegen würde, vergleiche Kapitel 3.2.3. Der Einsatz von Resident-Keys ermöglicht jedoch auch eine sichere *1FA* via FIDO2, wie nachfolgend erläutert.

Resident-Keys In den bisherigen Ausführungen wurde ein zustandsloser Authenticator mit serverseitiger Public-Key-Speicherung beschrieben („Server-side Credentials“ nach [HJL+21]). Der Vorteil liegt in der unbegrenzten Anzahl an Credentials je Authenticator. Nachteilig ist die Notwendigkeit zum Abruf der Credential-Daten vom Server. Resident-Keys, in [HJL+21] auch „Client-side discoverable Credentials“ genannt, lösen dieses Problem durch Credential-Speicherung auf dem Authenticator. Da der Speicherplatz des Security-Tokens limitiert

ist, können hier nur begrenzt viele Resident-Keys abgelegt werden. Bei Registrierung werden zum Resident-Key auf dem Authenticator $(key_{pri}, domain, fid, uid)$ sowie Metadaten, u. a. zum eingesetzten Public-Key-Verfahren, abgelegt.
Zur Anmeldung bei einer Anwendung mit der Domain *domain* sucht der Authenticator nach einem passenden Resident-Key und unterschreibt damit die Challenge des Servers. Sind mehrere Accounts zur Applikation hinterlegt, muss der Nutzer das passende Credential auswählen. Dieses Schema ermöglicht eine *1FA* ohne Angabe einer Nutzerkennung oder eines Passwortes. Sofern die *uid* nicht für die Fachlichkeit der Anwendung relevant ist, kann anstatt einer nutzergenerierten *uid* auch ein technischer Identifier, in [HJL+21] „userHandle“ genannt, vergeben werden, siehe hierzu Registraturformen in Kapitel 2.3.1.
Der Einsatz von Resident-Keys steigert die *UX*, da sowohl Registrierung als auch Anmeldung dann nur noch aus der Nutzerbestätigung am Authenticator bestehen müssen. Dann sollte allerdings eine *UV* und nicht nur eine *UP* erfolgen, damit eine Client-*2FA* (anstatt *1FA*) vorliegt, siehe Kapitel 4.2.1. Sie besteht dann aus den Merkmalen Besitz (der Authenticator) sowie Wissen (z. B. Codesperre am Authenticator) oder auch Inhärenz (z. B. Fingerabdruckleser am Authenticator).

Attestierung Im Kontext von FIDO2 kann die Echtheit des Authenticators beglaubigt werden, welche sich nach [Pow18] auf folgende Aspekte bezieht:

- Ein *MITM*, der in die FIDO2-Registrierung eingreift, kann den übermittelten Public-Key nicht durch seinen eigenen austauschen.
- Im Backend ist überprüfbar, ob der Authenticator dem angegebenen Hersteller sowie Modelltyp entspricht und notwendige Cipher-Suits unterstützt.

Um die Echtheitsprüfung zu bewerkstelligen, wird eine *PKI*-Architektur aufgespannt, siehe Kapitel 2.2.1. Jeder Authenticator erhält werkseitig ein statisches X.509-Zertifikat inklusive Schlüssel, das von einer Attestierungs-*CA* des Herstellers unterzeichnet wurde. Wenn die *IDP*-Anwendung über die Zertifikate der Attestierungs-*CAs* verfügt und ihnen vertraut, lassen sich die Attestierungszertifikate der Authenticators durch das Backend überprüfen. Ebenso kann eine Certificate-Revocation betrieben und bestimmten Herstellern oder Modellreihen das Vertrauen entzogen werden. Die Attestierung ermöglicht es, im Backend zu überprüfen, ob der Authenticator die Anforderungen zum gewünschten Sicherheitsniveau erfüllt, z. B. bezüglich unterstützter kryptographischer Verfahren. Zusätzlich hat die Applikation die Möglichkeit, via *WebAuthn* Sicherheitsvorgaben zum FIDO2-Prozess zu machen. So können z. B. *UV* anstatt *UP* eingefordert oder auch erlaubte *CTAP*-Transportprotokolle eingeschränkt werden.

Zur Attestierung muss der *IDP*-Betreiber eine Liste vertrauenswürdiger Attestierungs-*CAs* manuell pflegen. Ein weiterer Nachteil liegt in dem drohenden Privatsphäre-Problem für die Nutzer: Wird jedes Attestierungszertifikat mit einer eindeutigen Seriennummer versehen, so ist ein exaktes, anwendungsübergreifendes Tracking von Nutzern durch die Diensteanbieter möglich. Daher sollten Zertifikate je Hersteller oder Modelltyp vergeben werden.[Pow18] Allerdings sind Nutzer dann immer noch Kohorten zuordenbar, welche durch Kombination mit zusätzlichen Datenpunkten weiter verfeinerbar sind. Dem ist entgegen zu halten, dass die Attestierung als optionales Feature von FIDO2 nicht standardmäßig zum Einsatz kommt, sondern explizit durch die Webanwendung angefragt werden muss.

Deployment Abgesehen von der Attestierung vermeidet FIDO2 den Einsatz von Zertifikaten und operiert stattdessen direkt auf dem Schlüsselmaterial des Authenticators. Dadurch wird explizit ein Management von Zertifikaten durch den Nutzer vermieden. Für Privatpersonen vereinfacht dies potenziell die Handhabung des Security-Tokens und verbessert damit die *UX*. Eine Organisation bzw. Institution kann jedoch keine *PKI* für Authenticators direkt auf FIDO2 aufbauen. Ein Ausstellen und Hinterlegen von Nutzer-Zertifikaten durch die zuständige Stelle ist nicht vorgesehen. Nur wenn die einzusetzenden Authenticators Slots für Zertifikate bieten, ist eine *PKI*-Umsetzung möglich. Andernfalls können die in Kapitel 4.3.2 erläuterten Vorteile einer Distribution von Nutzer-Zertifikaten je Authenticator durch eine Zertifikats-*PKI* nicht ausgespielt werden. Stattdessen gilt es dann, einen Prozess zu etablieren, bei dem den Nutzern ihre unkonfigurierten Authenticators bereitgestellt werden und sie diese, wie oben erläutert, an der Anwendung registrieren müssen. Authenticator-Verwaltung sowie -Revocation können dann nicht via X.509-Standard mittels Zertifikaten erfolgen, sondern sind eigenständig durch die *IDP*-Applikation zu implementieren.

4.3.4 Risiko-Bewertung

Asymmetrische Merkmale entstammen prinzipbedingt der Kategorie Besitz. Damit können sie ihrem Eigentümer materiell entwendet werden. Das Risiko ist vor dem Hintergrund des Nutzerverhaltens sowie Verwendungskontextes zu bewerten und an dieser Stelle keiner allgemeinen Risikoeinschätzung zugänglich. Der Schlüsselbesitz sollte auf dem jeweiligen Gerät jedoch durch Faktoren der Kategorien Wissen oder Inhärenz lokal geschützt werden. Vom lokalen Schutz hängt dann die Sicherheit bei Entwendung des Faktors ab. In bestimmten Szenarien kann auch eine Zerstörung des Schlüsselmaterials nach zu vielen Fehlversuchen erfolgen.

Asymmetrische Verfahren bieten inhärent einen Replay-Schutz. Ein Angreifer könnte ihn als *MITM* umgehen, indem er die Kommunikation gezielt stört, sodass der Nutzer auf ein anderes Verfahren ohne Replay-Schutz ausweichen muss, siehe Tabelle 4.7. Für Webapplikationen könnte das z. B. *TOTP* sein, siehe Kapitel 4.2.4.
Im Falle eines *DB*-Leaks muss ein Angreifer aus den Public-Keys die Private-Keys zurückführen. Bei Einsatz geeigneter Cipher-Suits gestaltet sich der Angriff als nicht praktikabel, siehe Tabelle 4.8.

Tabelle 4.7: Risiko einer Replay-Attacke via Faktor-Downgrade

D	5/5	Impersonierung des Nutzers
R	3/5	Aufzeichnung und Wiedergabe einer Authentifizierung
E	1/5	Einbruch in geschützten *TLS*-Kanal
A	2/5	Betrifft jeweils einen Nutzer, parallelisierbar
$\sum$	**11/20**	⇒ **Mittleres Risiko**

Tabelle 4.8: Risiko eines DB-Leaks mit Public-Keys

D	4/5	Kenntnis eines *MFA*-Faktors
R	1/5	Erraten des Private-Keys
E	2/5	Zugang zu *DB*-Auszug
A	1/5	Schlüsselbestimmung je Nutzer erforderlich
$\sum$	**8/20**	⇒ **Niedriges Risiko**

Die in diesem Abschnitt vorgestellten Verfahren verhindern ein *Origin-Spoofing* und damit Passwort-Phishing bei Webapplikationen. Das gilt für die Authentifizierung mittels *MFA*-Faktoren, aber nicht zwangsweise für ihre Registrierung. Ein Nutzer könnte einen *MFA*-Faktor auf einer durch den Angreifer kontrollierten Webseite registrieren und die Applikation zukünftig über die Fake-Webseite verwenden. In diesem Szenario wäre kein Schutz gegen ein *Origin-Spoofing* gegeben. Einzig der Einsatz eines Security-Tokens, z. B. via FIDO2, schützt auch die Faktor-Registrierung. Hierzu lässt der *UA* den Domainnamen in den Prozess zur Schlüsselverarbeitung im isolierten Authenticator einfließen, sodass die Angabe im Backend überprüfbar wird.

Auch ohne materielle Entwendung können illegitime Zugriffe auf die Private-Keys erfolgen. Es gilt den Schutz privaten Schlüsselmaterials auf Ebene von Software und Hardware zu betrachten.
Der Private-Key kann ausschließlich im *Random-Access Memory (RAM)* gehalten werden, wie z. B. beim *TLS*-Token-Binding. Dann bedarf es eines privilegierten Prozesses zum Auslesen des Speichers oder eines Seitenkanals, siehe Tabelle 4.9. Wie in [GPTY18] aufgezeigt, könnte ein derartiger Seitenkanal gegen Caches der *Central Processing Unit (CPU)* oder spekulative Code-Execution auch von

einer unprivilegierten Webseite stammen. Anstatt im flüchtigen *RAM* müssen Private-Keys mitunter auch persistent auf der Festplatte vorgehalten werden, z. B. für *mTLS* als Zertifikatsdatei oder im *UA*-Speicher für das Verfahren nach Kapitel 4.3.1. Das Auslesen oder Exfiltrieren von dauerhaft gespeicherten, unverschlüsselten Private-Keys ist für bösartige Programme auf dem gleichen System mit weniger Berechtigungen bzw. Hürden verbunden, siehe Tabelle 4.10.

Tabelle 4.9: Risiko zur Schlüsselextraktion aus dem RAM

D	4/5	Kenntnis von *MFA*-Faktoren
R	1/5	Privilegierter Prozess oder Seitenkanal benötigt
E	2/5	Lokale Ausführung auf dem Endgerät des Opfers
A	2/5	Betrifft jeweils einen Nutzer, parallelisierbar
$\sum$	**9/20**	$\Rightarrow$ **Niedriges Risiko**

Tabelle 4.10: Risiko zur Schlüsselextraktion von der Festplatte

D	4/5	Kenntnis von *MFA*-Faktoren
R	5/5	Finden und Auslesen von Schlüsseldateien
E	2/5	Lokale Ausführung auf dem Endgerät des Opfers
A	2/5	Betrifft jeweils einen Nutzer, parallelisierbar
$\sum$	**13/20**	$\Rightarrow$ **Mittleres Risiko**

Auf Security-Tokens lässt sich der Angriff zur Schlüsselextraktion mittels Software nicht übertragen, da der Ausführungskontext von Schlüsselverarbeitung und Applikationssoftware durch die Gerätebarriere getrennt ist. Die *FIDO*-Allianz hat in [Lin21] mögliche Angriffsvektoren für das FIDO2-Protokoll analysiert. Einem Threat-Modelling sind die nachfolgend erläuterten zuzuführen.

Tabelle 4.11: Risiko von FIDO2 bei kompromittiertem Client ohne Token Binding

D	5/5	Anmeldung durch Angreifer am Account
R	4/5	Versenden und Nutzen abgefangener Werte
E	2/5	Kompromittierung der Client-Software
A	4/5	Aktive Nutzer erhalten kompromittierten Client
$\sum$	**15/20**	$\Rightarrow$ **Hohes Risiko**

Tabelle 4.12: Risiko von FIDO2 bei kompromittiertem Client mit Token-Binding

D	5/5	Anmeldung durch Angreifer am Account
R	1/5	Extraktion des Private-Keys vom *TLS*-Kanal
E	1/5	Eindringen in TLS-Kanal, Auslesen geschützer *RAM*
A	1/5	Opfer kompromittierter *TLS*-Verbindungen
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Ein Desktop-Programm ist im Gegensatz zu einem browserbasierten Client nicht der Origin-Bindung unterworfen. Damit könnte eine Desktop-Anwendung das FIDO2-Protokoll anstatt für die eigene für eine beliebige Origin durchführen,

siehe Tabelle 4.13. Als Mitigation dienen spezielle Security-Tokens, die über ein Display die angeforderte Aktion inklusive Origin anzeigen können.

Ein weiterer Threat betrifft die Kompromittierung der Client-Software: Ein Angreifer könnte z. B. in den Webclient bösartiges *JavaScript* injizieren, um darüber FIDO2-Authentifizierungen mitzuschneiden und selbst zu verwenden, siehe Tabelle 4.11. Als Mitigation kann das in Kapitel 4.3.1 erläuterte Token-Binding eingesetzt werden, dessen Verwendung auch der FIDO2-Standard als optionales Feature unterstützt.[HJL+21] Dadurch werden die FIDO2-Authentifizierungsnachrichten an den *TLS*-Kanal gebunden.
Ein Angreifer müsste zur Umgehung den Private-Key des Token-Bindings zur Laufzeit aus dem Speicherbereich der eingebundenen *TLS*-Bibliothek extrahieren. Andernfalls müsste er als *MITM* bereits in den *TLS*-Verbindungsaufbau eingegriffen haben, um einen eigenen Private-Key zum Token-Binding zu nutzen. Tabelle 4.12 zeigt die Risiko-Betrachtung zur Umgehung des Token-Bindings.

Tabelle 4.13: Risiko zum Origin-Spoofing mit FIDO2 via Desktop-Anwendung

D	5/5	Aushebelung FIDO2-Faktoren für beliebige Origins
R	4/5	Verwendung von FIDO2 in Desktop-Anwendung
E	3/5	Installation Trojaner auf dem Opfer-Gerät
A	2/5	Betrifft jeweils einen Nutzer, parallelisierbar
$\sum$	**14/20**	$\Rightarrow$ **Mittleres Risiko**

Tabelle 4.14: Risiko zum Brute-Force-Angriff auf Authenticators ohne UP

D	3/5	Möglichkeit zum Erraten eines Faktors
R	4/5	Brute Force gegen Authenticator
E	2/5	Permanenter Zugang zum Authenticator
A	2/5	Betrifft jeweils ein System, parallelisierbar
$\sum$	**11/20**	$\Rightarrow$ **Mittleres Risiko**

Wenn ein Angreifer die *UP*-Prüfung durch den Security-Token umgehen kann, verfügt er über eine Brute-Force-Möglichkeit. Solange der Security-Token mit dem Nutzer-Gerät verbunden ist, ließe sich der Angriff durchführen. Insbesondere *TPMs* sind durchgängig mit dem Computer verbunden und bieten keine *UP*. Tabelle 4.14 zeigt eine Risiko-Betrachtung hierzu.

Ein Angreifer von Innen, der in die Herstellung der Security-Tokens involviert ist, könnte Backdoors in der Firmware platzieren oder Authenticators ähnlich beeinträchtigen, siehe Tabelle 4.15. Der Nutzer muss dem Hersteller hierbei letztendlich vertrauen.

Die in Kapitel 4.3.3 erläuterte Attestierung birgt das Privatsphäre-Problem, welches durch eine große Kohortengröße verhindert werden soll. Ob diese Kohor-

tengröße tatsächlich erzielt wird, ist für den Nutzer nicht nachvollziehbar. Im Zweifelsfall muss ein Nutzer annehmen, das einzige Mitglied einer Kohorte zu sein. Ein Nutzer-Tracking wäre dann anwendungsübergreifend möglich, siehe Tabelle 4.16.

Tabelle 4.15: Risiko von durch den Hersteller kompromittierten Security-Tokens

D	5/5	Aushebelung FIDO2-Faktoren für beliebige Anwendungen
R	1/5	Manipulation an der Token-Firmware
E	1/5	Zugang zum Unternehmensnetz
A	2/5	Nutzer von Tokens des Herstellers
$\sum$	**9/20**	$\Rightarrow$ **Niedriges Risiko**

Tabelle 4.16: Risiko zum Nutzer-Tracking via präparierter Security-Tokens

D	2/5	Verknüpfung von Accounts verschiedener Anwendungen
R	2/5	Aggregation attestierter FIDO2-Verbindungen
E	2/5	Attestierungszertifikat für kleine Kohorten ausstellen
A	2/5	Nutzer eines betroffenen Tokens, parallelisierbar
$\sum$	**8/20**	$\Rightarrow$ **Niedriges Risiko**

Beurteilung Der Einsatz von Security-Tokens in Kombination mit *TLS*-Token-Binding bietet von den untersuchten Verfahren die besten Sicherheitseigenschaften. Die diesbezüglich aufgezeigten Threats wurden als wenig bedrohlich eingestuft. Einzig der Einsatz von bösartigen Desktop-Programmen oder ähnlich privilegierter Software kann Accounts gefährden, die mittels Security-Token abgesichert sind. Allerdings hätte ein Angreifer in dieser Position auch mächtigere Möglichkeiten, andere Programme mit Malware zu infizieren.
Security-Tokens mit *UP*-Überprüfung sind als starke *MFA*-Faktoren zu betrachten. Zusätzlich zum Besitz sollte eine iterative *MFA* obligatorisch ein Wissensmerkmal beinhalten. Alternativ lässt sich allein mittels Security-Tokens eine starke, nicht-iterative Authentifizierung erzielen, sofern der Token via *UV*-Prüfung vor unbefugtem Zugriff geschützt wird. Für die sich hieraus ergebende Client-*2FA* können wiederum Merkmale der Kategorie Wissen oder Inhärenz genutzt werden. Für den Fall des Token-Verlustes sind Alternativen vorzuhalten, z. B. weitere zum Account registrierte Security-Tokens.
Eine Schlüsselablage auf den Endgeräten der Anwender ist weniger sicher als die Nutzung eines Security-Tokens. Jedoch bietet dieser Ansatz eine bessere *UX* und ist für Anwendungen ohne hohe Kritikalität dennoch geeignet. Dies gilt insbesondere, wenn Nutzer andernfalls nur auf eine *1FA* via Passwort zur Accountsicherung zurückgreifen würden.

5 Realisierung

Die Realisierung umfasst die prototypische Implementierung der Verfahren inklusive Evaluation mittels Tests. Abschließend wird die Umsetzung bewertet.

5.1 Implementierung

Es folgt die Erläuterung von Aspekten zur Realisierung, die noch nicht oder unvollständig aus den bisherigen, theoretischen Ausführungen hervorgehen. Der Quellcode ist in Anhang D (Digitaler Anhang) zu finden.

5.1.1 Umgesetzte Verfahren

Die Referenz-Implementierung basiert auf dem Protokoll aus Kapitel 2.3. Aus Kapitel 3 und 4 werden die folgenden Konzepte übernommen:

- Clientseitige Überprüfung der Stärke bei Wahl neuer Passwörter anhand von Regeln sowie Abgleich mit schwachen Passwörtern via Server-Dienst ohne Passwort-Bekanntgabe an das Backend, entsprechend Kapitel 3.1.
- Langsames, clientseitiges Passwort-Hashing mit Salt ohne Pepper, entsprechend Kapitel 3.2.4.
- Hashing mit Server-Relief via *SCRAM* inklusive serverseitiger Hash-Verschlüsselung als Peppering-Ersatz, entsprechend Kapitel 3.3.3.
- Einsatz von *SCRAM* mit *SCRAM*-Tokens und ohne *TLS* Channel oder Token Binding zur Vermeidung von Layer-Abhängigkeiten, entsprechend Kapitel 3.3.3 und 4.3.1.
- Möglichkeit zur Rotation der Parameter zum clientseitigen Hashing (Hashing-Algorithmus, Cost sowie ggf. Salts), entsprechend Kapitel 3.2.2.
- Optionaler, iterativer *MFA*-Prozess, entsprechend Kapitel 4.1.2.
- Bereitstellung eines Backup-Keys für die *MFA*, entsprechend Kapitel 4.1.4.
- Nutzer-gesteuerte Verwaltung der *MFA*-Faktoren und Konfiguration des *MFA*-Levels je Account, entsprechend Kapitel 4.1.4.
- *MFA*-Faktoren via FIDO2 bzw. WebAuthn, entsprechend Kapitel 4.3.3.
- *UA*-Keys als *MFA*-Faktor für „Trusted Devices“, entsprechend Kapitel 4.3.1.
- *MFA*-Faktoren via *TOTP*-Verfahren als Ausweichoption für rein browserbasierte Clients, entsprechend Kapitel 4.2.2.

Der Prototyp in Anhang D unterstützt die Betriebsmodi Client-Hashing oder Server-Hashing. Letzteres wird nur für den Vergleichstest in Kapitel 5.2.3 benötigt und ist kein Teil der weiteren Betrachtung.

5.1.2 Datenmodell & Technologien

Die Implementierung der Verfahren bedarf einer Datenspeicherung im Backend. Das zugrundeliegende Datenmodell zeigt Abbildung 5.1 als *Entity Relationship Model (ERM)*. Die Attribute besitzen einen persistenten oder volatilen Charakter, der zu einer getrennten Datenhaltung führt:

- **Persistente Attribute** sind zur langfristigen Speicherung vorgesehen. Die feste Strukturierung mittels Felddefinitionen sichert die Integrität des Datenbestandes auch bei Änderungen an Schema oder Inhalt. Als Software-Komponente kommt das relationale *DB*-System *PostgreSQL* zum Einsatz.
- **Volatile Attribute** werden nur innerhalb einer begrenzten Lebensdauer benötigt. In dieser sind sie besonders relevant und müssen äußerst schnell abrufbar sein. Da es sich um kompakte Datenstrukturen handelt, ist eine Leistungsoptimierung anwendbar: Anstatt die volatilen zu den persistenten Werten auf die Festplatte zu schreiben, werden sie im *RAM* gehalten. Der *TMTO* wird zugunsten verkürzter Zugriffszeiten gegenüber dem zusätzlichen *RAM*-Bedarf entschieden. Als Software-Komponente kommt die In-Memory-*DB Redis* zum Einsatz.

Die Verwendung fortlaufend inkrementierter Ganzzahlen als Identifier ist für die vorliegende Applikation zu vermeiden. Ein Angreifer könnte sonst Nutzungsdaten über das System sammeln, z. B. wie viele *MFA*-Faktoren in einem gewissen Zeitraum registriert wurden. Stattdessen sieht das Datenmodell den Einsatz von *Universally Unique Identifiers (UUIDs)* in Version 4 vor, welche als 128-Bit Zufallszahlen erzeugt werden. Eine Kollision von Identifiern ist damit hinreichend unwahrscheinlich, insbesondere bei zusammengesetzten Primärschlüsseln.

Zusätzlich zur Datenhaltung wird eine applikationsübergreifende Konfiguration benötigt, dargestellt in Abbildung 5.2. Diese enthält Grenzwerte für die eingesetzten Techniken sowie Deployment-Angaben. Außerdem beinhaltet sie Vorgabewerte für die Parameter der Verfahren zur Authentifizierung. Bei Registrierung eines neuen Faktors am System werden die Parameter der Konfiguration entnommen und in der *DB* hinterlegt. Dadurch können die Vorgaben aktualisiert werden, ohne die Benutzbarkeit bestehender Faktoren zwangsweise einzuschränken.

Neben der Datenhaltung im Backend erfolgt ein Zustandstransport zwischen Client und Server. Dies geschieht mittels Tokens, die Tabelle 5.1 als Übersicht zusammenfasst. Zusätzlich dokumentiert Anhang B die Datenverarbeitung und Nutzungsabläufe des prototypischen Webclients anhand von Screenshots.

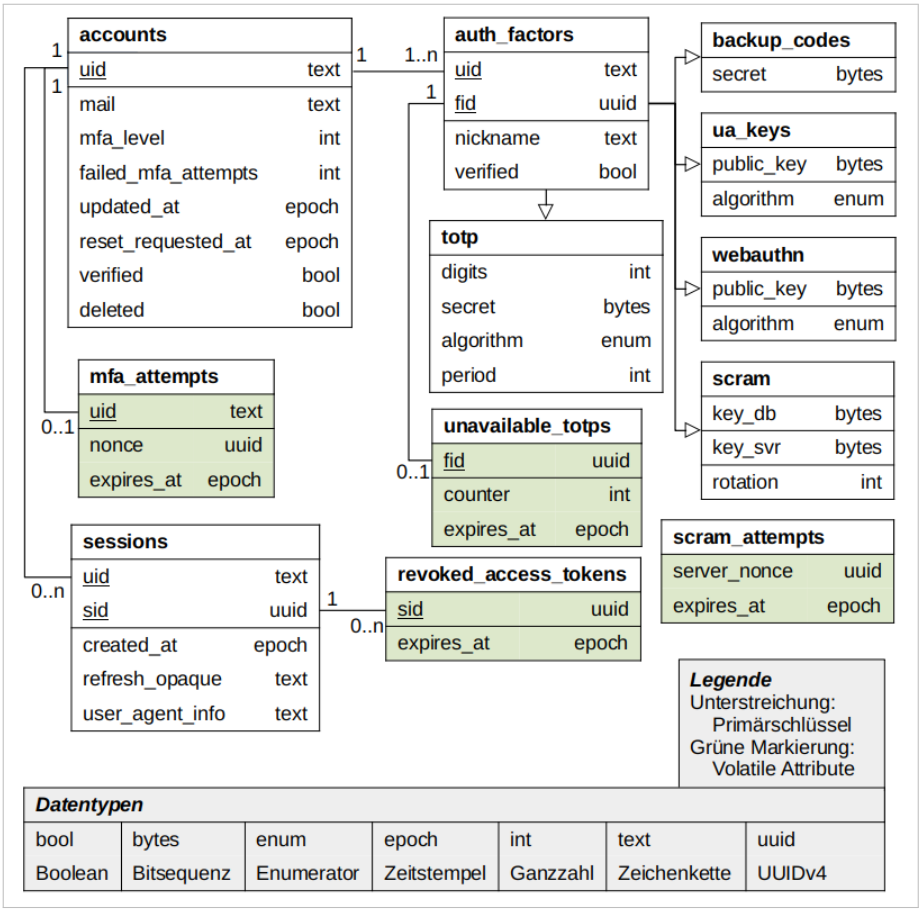


Abbildung 5.1: *ERM* der Datenhaltung zur Referenz-Implementierung
Quelle: Eigene Darstellung

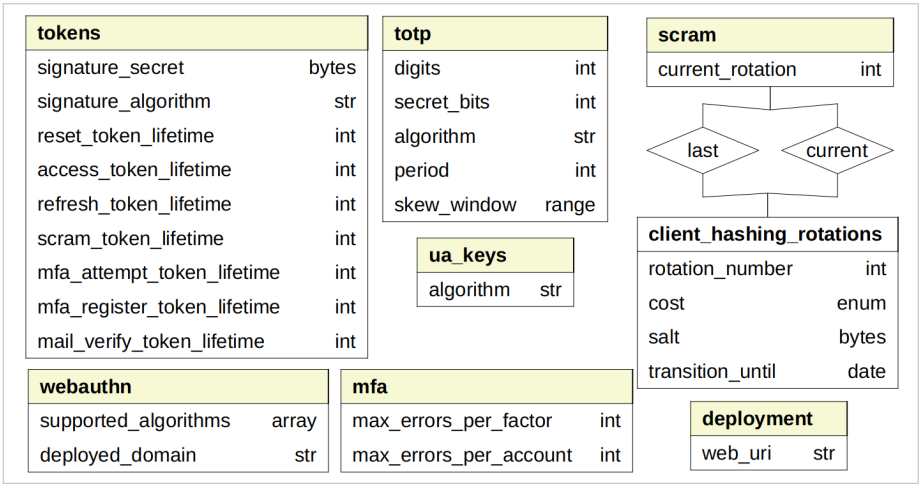


Abbildung 5.2: *ERM* der essenziellen Konfiguration zur Referenz-Implementierung
Farblich hervorgehoben sind *Singletons*. Quelle: Eigene Darstellung

Tabelle 5.1: Übersicht zu eingesetzten Tokens und deren Eigenschaften

Token Funktion	Kapitel Referenz	Typ	Kontext	Abhängigkeiten zur Verwendung	Attribut zur Einmaligkeit	Invalidierung
Access	2.3.3	client	Session	Abschluss MFA	`tid`	via TRL
Refresh	2.3.3	server	Session	Besitz abgelaufener Access Token	`tid`	Löschung aus der DB
Mail Verifik. zur Registrierung	2.3.1, 4.1.4	client	Regis-trierung	Account noch nicht verifiziert	/	Abschluss Registrierung
Mail Verifik. zur Accountlöschung	2.3.6, 4.1.4	client	Löschung	Gültige Session	/	Abschluss Löschung
Mail Verifik. zur MFA Konfiguration	4.1.4	client	MFA Konfigu-ration	Gültige Session	/	Übernahme der Konfiguration
Mail Verifik. zur Zurücksetzung des Passwortes	2.3.4, 4.2.3	client	Zurück-setzung	Angabe Mail-Adresse	`reset_requested_at`	Passwort-wechsel
MFA Login Attempt	4.1.2, 4.1.4	client	Iterative MFA	Abschluss 1FA	`mfa_attempt_id`	Löschung MFA Attempt vom Server
Mail Verifik. zur MFA Faktor Registrierung	4.1.4	client	MFA Konfigu-ration	Gültige Session	`fid`	Abschluss Faktor-Registrierung
SCRAM Token	3.3.3	client	Login via 1FA	/	`scram_server_nonce`	Löschung Nonce vom Server

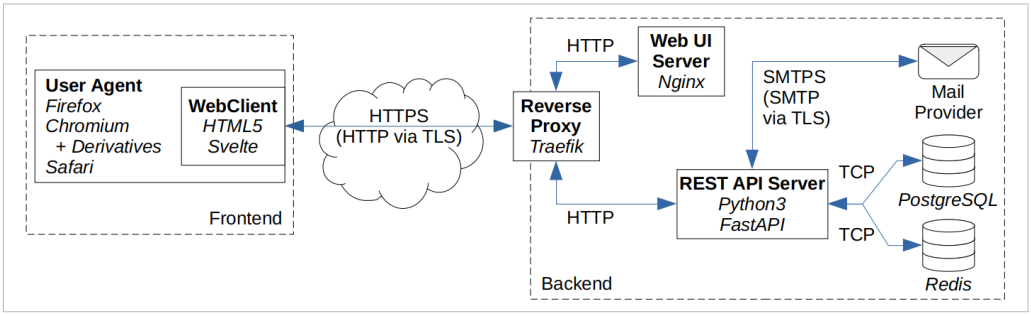


Abbildung 5.3: Architektur und Technologien der Referenz-Implementierung
Kursivschrift kennzeichnet Technologien. Quelle: Eigene Darstellung

Das Architektur-Modell basiert auf den Überlegungen aus Kapitel 2.1.1. Sein Aufbau inklusive eingesetzter Technologien wird in Abbildung 5.3 gezeigt:

Datenhaltung: Der Einsatz der Datenbanksysteme *PostgreSQL* und *Redis* ist bereits am Anfang des Abschnitts erläutert worden.

Mail-Versand: Der Mail-Provider erlaubt es, E-Mails via *Simple Mail Transfer Protocol (SMTP)* zu verschicken. Befindet er sich als externer Dienst außerhalb des geschützten Server-Netzwerkes, so gilt es wie dargestellt, auch diese Kommunikationsstrecke durch *TLS* zu schützen.

Webserver: *Traefik* und *Nginx* sind etablierte Software-Projekte, die als Reverse-Proxy sowie zur Auslieferung der statischen Dateien des Webclients zum Einsatz kommen.

Zur Implementierung der Applikationslogik kommt die schwach-typisierte, speichersichere Skript-Sprache *Python* zum Einsatz, welche ein schnelles Prototyping ermöglicht. Da die *Python*-Runtime Programme interpretiert, ist die Ausführung prinzipbedingt langsamer als bei kompilierten Sprachen. Eine sichere Implementierung zum Produktiveinsatz sollte daher in einer kompilierten, stark-typisierten sowie speichersicheren Sprache erfolgen. Wie Python sollte diese asynchrone Programmierung ermöglichen und darüber hinaus zur Performance-Optimierung echte Nebenläufigkeit sowie native Mehrprozessor-Unterstützung bieten, wie dies z. B. bei den Sprachen Go und Rust der Fall ist.

Die Programm-*API* nutzt *HTTP* als Protokoll und orientiert sich dabei am *Representational State Transfer (REST)*-Paradigma. Als *Python*-Framework zur Umsetzung der *REST-API* kommt FastAPI zum Einsatz, welches auch die Daten-Validierung und -Serialisierung übernimmt. Die Definition der *REST-API* für den implementierten Prototyp ist in Anhang C dokumentiert.

Die Server-Programme werden auf Linux-Systemen ausgeführt. Als *CSPRNG* wird die Ausgabe der Gerätedatei `/dev/random` des Linux-Kernels genutzt.

Deployment: Die einzelnen Backend-Komponenten werden als Software-Container paketiert, wozu die Software *Docker* dient. Dies ermöglicht einen skalierbaren, isolierten und reproduzierbaren Betrieb in sterilen Umgebungen.

Webclient: Die angedachte Ausführung des Webclients durch moderne Webbrowser vermeidet eine Kopplung an die Client-Plattform. Entsprechend kommen die Technologien der Browser-*Sandbox* nach *HTML5*-Standard zur Anwendung. Von diesen sind insbesondere folgende zu nennen:

- **IndexedDB**: Daten-Persistierung im *UA*-Speicher mit Origin-Isolation.
- **Fetch API**: Datenaustausch mit dem Backend im Kontext einer Webseite.
- **Web Cryptography API**: Ausführung kryptografischer Operationen.
- **Web Authentication API**: Einbindung von FIDO2, siehe Kapitel 4.3.3.

Zusätzlich wird *Svelte* als Web-Framework eingesetzt. Es ermöglicht die Entwicklung vollwertiger, reaktiver und in Komponenten organisierter Client-Anwendungen als Webseiten. Diese saubere Strukturierung erhöht nicht nur die Erweiterbarkeit und Wartbarkeit, sondern senkt auch die Komplexität der Implementierung. Dadurch können präventiv Bugs an sicherheitsrelevanten Quellcode-Passagen besser vermieden werden.
Es entsteht eine *Single-Page Application (SPA)*, welche sämtliche clientseitige Applikationslogik eigenständig bündelt und als Dateienpaket bereitstellt. Diese Eigenschaft ist für die nachfolgende Sicherheitsbetrachtung relevant.

5.1.3 Sicherheit nativer und browserbasierter Applikationen

Als Vorteil clientseitiger Passwortverarbeitung wurde aufgeführt, dass der Server keine Passwort-Klartexte erfährt. Wird die Server-Infrastruktur kompromittiert, so könnte jedoch ein modifiziertes Client-Programm zum Download angeboten werden. Dieses könnte unbemerkt die Credentials ausleiten. Es gilt Auswirkungen einer Server-Kompromittierung auf die Client-Software zu untersuchen.

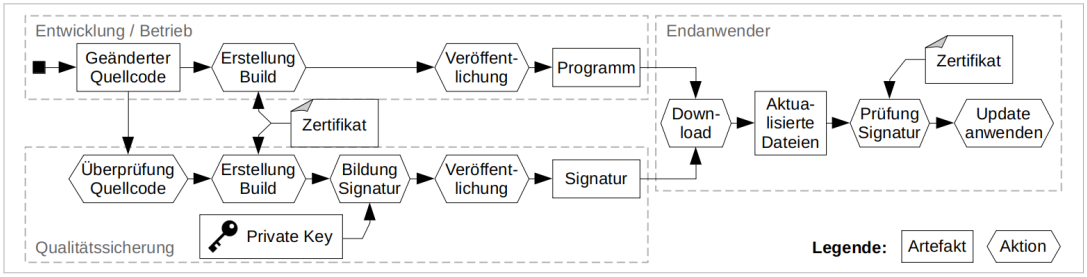
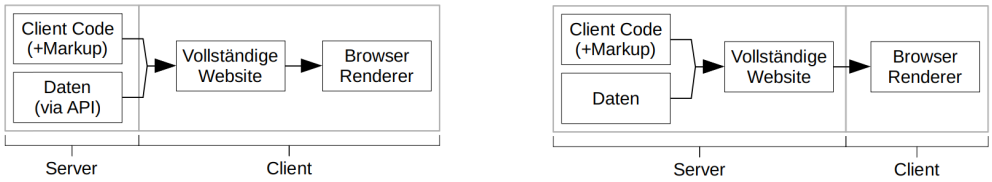


Abbildung 5.4: Ablauf eines signaturgestützten Prozesses zur Software-Aktualisierung
Quelle: Eigene Darstellung

Desktop-Programme sind bei Server-Kompromittierung anfällig für bösartige Updates. Da sie den Kontext ihrer Ausführung kontrollieren, kann der Update-Prozess über eine Signaturprüfung abgesichert werden. Dazu sind organisatorische Maßnahmen zur getrennten Erstellung von Programm und Signatur nach Vier-Augen-Prinzip zu ergreifen. Abbildung 5.4 zeigt einen exemplarischen, zertifikatsbasierten Prozess. Dieser ist abstrahiert auch zur signierten Distribution paketierter Software dienlich. Ein Sicherheitsvorteil entsteht jedoch nur, wenn der Signatur-Aussteller vor Signierung jede Änderung am Quellcode kritisch prüft. Falls dieser Aufwand nicht konsequent betrieben wird, erzeugt das Verfahren lediglich ein falsches Sicherheitsgefühl. Des Weiteren funktioniert die Signaturprüfung nur, wenn der Nutzer über den richtigen Public-Key, respektive Zertifikat, verfügt. Demnach muss der Endanwender der Quelle zum erstmaligen Beziehen der Software vertrauen, was dem Sicherheitskonzept *Trust On First Use (TOFU)*

entspricht. *TOFU* wird z. B. auch von *SSH* zur Überprüfung der Server-Identität nach Verbindungsaufbau genutzt, falls noch kein sog. Host-Key etabliert wurde.



(a) Client-Rendering **(b)** Server-Rendering

Abbildung 5.5: Vergleich von Rendering-Typen für browsergestützte Applikationen
Quelle: Eigene Darstellung

Der Signaturprozess ist nicht direkt auf browserbasierte Applikationen übertragbar: Diese können nicht über ihren Ausführungskontext (Webbrowser) bestimmen und folgen dem Request-Response-Modell. Hierbei sind die Architektur-Typen Client-Rendering und Server-Rendering unterscheidbar, siehe Abbildung 5.5. Da beim Server-Rendering für jeden Request eine individuelle Response generiert wird, ist die Signierung nicht sinnvoll anbringbar. Zum Abgreifen der Credentials reicht einmalig eine manipulierte Response. Ohne Aufzeichnung und Auswertung der Kommunikation ist diese Manipulation clientseitig weder nachweisbar noch reproduzierbar.

Im Gegensatz dazu ist Client-Rendering mit dem Signatur-Prozess kompatibel, da es funktional ähnlich zu Desktop-Programmen einsetzbar ist: Der Client-Quellcode wird als *SPA* einmalig vollständig vom Server bezogen, siehe vorheriger Abschnitt. Die Browser-Technologie der *Progressive Web Apps (PWAs)* erlaubt es, den Code dauerhaft am Client vorzuhalten. Über den Browser-Mechanismus der Service-Worker kann die Webseite einen transparenten *HTTP*-Proxy zur Server-Kommunikation etablieren. Dadurch könnte der Client eine Signatur-Prüfung durchführen, bevor neue Quellcode-Dateien ausgeführt werden.

Allerdings sieht das Vertrauensmodell des Webs vor, dass der Server im Zweifelsfall immer den Client-Quellcode überschreiben kann, z. B. durch Setzen des *HTTP*-Response-Headers `Clear-Site-Data`. Dies ist aus *UX*-Perspektive sinnvoll, da ein Bug die Client-Anwendung sonst irreparabel in einen dysfunktionalen Zustand versetzen könnte.

Solange Web-Technologien keine *API* zur durchsetzbaren Code-Signierung anbieten, muss die Distribution hierfür außerhalb dieses Vertrauensmodells erfolgen. Beispielsweise wäre der Client-Code als Browser-Extension, Desktop-Programm oder mobile App paketierbar. Erfolgt die Distribution der paketierten Software über einen Drittanbieter, so muss anstatt des Browser-Herstellers und -Distributors dem Drittanbieter vertraut werden, z. B. einem App Store.

Eine Server-Kompromittierung wirkt sich auch direkt auf browser-gestützte Webclients aus. Der Vorteil von Client-Hashing liegt in diesem Szenario demnach lediglich im *Server-Relief* und Schutz vor passiven Eavesdroppern.

5.1.4 Datengrundlage zur Feststellung schwacher Passwörter

Die Software wurde als in sich abgeschlossen konzipiert. Um die Überprüfung der Passwortstärke aus Kapitel 3.1 durchführen zu können, benötigt die Applikation entsprechend eine Liste von Hashwerten schwacher Passwörter. Hierzu wird exemplarisch der Datensatz des Projektes „Have I Been Pwned“ mit Stand vom 19. November 2020 verwendet.[1]
Der Datensatz wurde aus Passwörtern bekannter Data-Breaches gebildet. [Hun18] Er umfasst 613.584.246 einmalige, mittels *SHA*-1 gehashte Passwörter sowie dazu jeweils die Anzahl der Vorkommnisse in den Data-Breaches. Beispielsweise ist „123456“ das am häufigsten vorkommende Passwort, siehe Listing 5.1.

```
1  7C4A8D09CA3762AF61E59520943DC26494F8941B:24230577    → 123456
2  B1B3773A05C0ED0176787A4F1574FF0075F7521E:3993346     → qwerty
3  5BAA61E4C9B93F3F0682250B6CF8331B7EE68FD8:3861493     → password
4  7A3BEC69F4A530B94A02B55D93C717F9AC9D4F18:105         → Wings123!
```

Listing 5.1: Beispieleinträge aus dem Datensatz von „Have I Been Pwned“ ergänzt um die Passwort-Klartexte

Im Rohformat als Textdatei mit einer unkomprimierten Größe von $25,2\,GiB$ ist der Datensatz schlecht verarbeitbar. Durch Entfernung der nicht benötigten Vorkommnis-Anzahlen und binäre Speicherung wird die Größe auf $11,4\,GiB$ gesenkt. In dieser kompakten Speicherform werden je Eintrag nur die $160\,Bits$ bzw. $20\,Byte$ eines *SHA*-1-Hashwertes benötigt.
Schwache Passwörter, die bereits durch die Passwortregeln unterbunden werden, könnten zur weiteren Reduzierung von der Liste getilgt werden. Allerdings würde die Liste durch Kopplung mit den Passwortregeln weniger generisch einsetzbar.

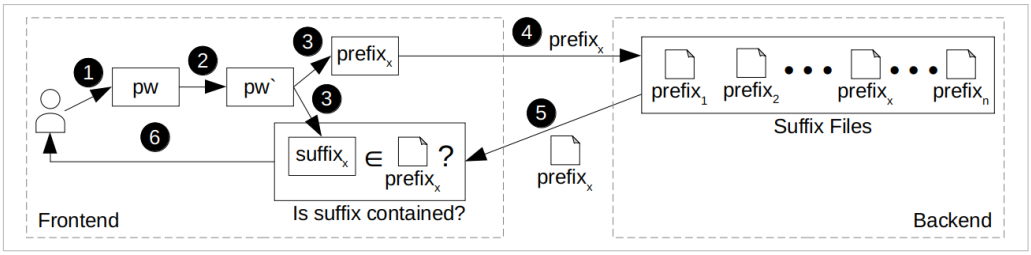


Abbildung 5.6: Ablauf der Überprüfung zur Passwortstärke ohne Passwort-Exponierung an den Server mithilfe des Datensatzes von „Have I Been Pwned“ Quelle: Eigene Darstellung

[1] Der Datensatz ist unter `https://haveibeenpwned.com/Passwords` beziehbar.

Tabelle 5.2: Möglichkeiten zur Wahl der Präfixlänge zur k-Anonymität
Hervorgehoben ist die schlussendlich gewählte Präfixlänge

	Bitanzahl	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Präfixe	**Hex-Zeichen (Nibbles)**	3	4	4	4	4	5	5	5	5
Speicherbedarf	**Je Hashwert (in Bytes)**	19	19	19	19	18	18	18	18	18
	Gesamt (in GiB)	10,9	10,9	10,9	10,9	10,3	10,3	10,3	10,3	10,3
	Ersparnis	5%	5%	5%	5%	10%	10%	10%	10%	10%
Übertragung	**Ø Antwort Hashwerte (in Tausend)**	150	75	37	19	9	5	2	1	1
	Ø Antwortgröße (in KiB)	2780	1390	695	347	165	82	41	21	10
	Hashanteil	8%	8%	9%	9%	10%	11%	11%	12%	13%

Das Verfahren aus Kapitel 3.1 bedarf einer Übertragung von Hashwert-Präfixen, wobei für eine starke k-Anonymität möglichst wenig Hashbits zu übertragen sind. Der Datensatz wurde nach Präfixen gruppiert und die Hashwerte danach unterteilt. Je Präfix entsteht eine Datei mit zugehörigen Hash-Suffixen, die bei Anfrage des entsprechenden Präfixes zurückgegeben werden. Abbildung 5.6 veranschaulicht das Verfahren.

Aufgrund des exponentiellen Zusammenhangs zwischen Hashbit-Anzahl und Antwortgröße gilt es, praktikable Werte zu finden. Anhand der in Tabelle 5.2 dargestellten Überlegung wird eine Präfixlänge von 16 Bit gewählt: Da in den Hash-Dateien eine Speicherung des Präfixes entfällt, wird die Speichermenge je Hash auf $20-2 = 18\,Bytes$ gesenkt. Aufgrund des Lawineneffektes gilt eine annähernde Gleichverteilung der Hashwert-Anzahl je Präfix, sodass mit Durchschnittsgrößen für Hashwert-Anzahlen gearbeitet werden kann. Eine durchschnittliche Antwortgröße von $165\,KiB$ wird als schnell übertragbar und eine Antwortmenge von 9 000 Einträgen als leicht verarbeitbar angenommen. Die Präfixe werden als hexadezimale Zeichen enkodiert, sodass jedes Zeichen ein Nibble (Halbbyte) ausdrückt. Bei $16\,Bits$ werden vier Nibbles exakt ausgefüllt, sodass keine Bit-Operationen zur Präfixbildung oder Auswertung der Antwort-Hashsuffixe notwendig sind. Es entstehen $2^{16} = 65\,536$ mögliche Präfixe. Wie in Kapitel 3.1.2 geschildert, offenbart das Verfahren dem Server nicht mehr Information über das Passwort als die Zugehörigkeit zu einem Präfix.

Damit ein *MITM* aus der *TLS*-verschlüsselt übertragenen Antwort-Datenmenge (Schritt 5 in Abbildung 5.6) nicht auf das angeforderte Präfix schließen kann, ist

die Server-Antwort um ein Padding zu ergänzen. Bei 16 *Bits* Präfixlänge liegt die Antwortgröße für den Datensatz zwischen 156 *KiB* und 173 *KiB*. Durch das Padding werden die Antworten mit Zufallsdaten auf den größtmöglichen Wert (173 *KiB*) verlängert. Über einen zusätzlichen *HTTP*-Header wird die reale Größe übermittelt, sodass der Client das Padding entfernen kann.

Die vorgestellte Funktion zur Feststellung schwacher Passwörter operiert unabhängig zur übrigen Applikationslogik. Die Server-Skalierbarkeit wird aufgrund der Größe des Hash-Datensatzes äußerst negativ beeinflusst. Deshalb ist es empfehlenswert, die Funktion abgekoppelt als eigenständigen *Micro-Service* zu betreiben und separat von anderen Backend-Komponenten zu skalieren.

5.1.5 Rate-Limiting

Auf die Sicherheit durch Rate-Limiting wurde als Querschnittsthema an mehreren Stellen verwiesen. Deshalb folgt ein Überblick zu Möglichkeiten und Grenzen.

Ohne Rate-Limiting kann ein Angreifer mit der höchstmöglichen Datendurchsatzrate so viele Anfragen an den Server stellen, wie technisch möglich. Überschreitet die Server-Last aufgrund des Anfrageaufkommens die skalierbaren Server-Kapazitäten, so wird die Verfügbarkeit der Applikation eingeschränkt oder unterbunden. Über eine *DDOS*-Attacke ist dieser Systemausfall gezielt provozierbar.
Als Gegenmaßnahme kann die durch einen Request maximal erzeugbare Serverlast reduziert werden. Beim Server-Hashing betrifft dies die langsamen Operationen zum Passwort-Hashing. Wie in Kapitel 3.2.4 herausgestellt, ist Client-Hashing als architektonische Maßnahme zur Verkleinerung dieses Angriffsvektors nützlich. Dadurch wird der Hashing-Aufwand zur Durchführung einer Brute-Force-Attacke zum Ausprobieren möglicher Passwort-Kandidaten vom angegriffenen Server-System zum Angreifer verschoben.
Allerdings verhindert Client-Hashing nicht den Bedrohungsvektor durch Brute-Force-Attacken von Angreifern mit hinreichenden Hashing-Kapazitäten. Da das Server-System nun noch mehr Anfragen je Zeiteinheit verarbeiten kann, wird ein Brute-Force mit Passwort-Kandidaten sogar effektiver. Deshalb muss der Server Anfragesteller nach Kriterien unterteilen und eine obere Schranke für das geduldete Anfrageaufkommen je Zeitfenster bestimmen. Für die verschiedenen Server-Endpunkte können unterschiedlich strenge Rate-Limits sinnvoll sein. Beispielsweise ist ein Passwort als Parameter deutlich leichter zu erraten als die korrekte Signatur für einen gefälschten Bearer-Token. Wird ein Rate-Limit überschritten, so ist die Anfrage abzuweisen. Alternativ sind weitere Nachweise zur Legitimität des Anfragestellers zu erbringen, wie am Ende des Abschnitts dargelegt.

Etablierung eines Rate-Limitings

Für angemeldete Nutzer ist ein Rate-Limiting über die *uid* realisierbar. Die Schwierigkeit besteht im Umgang mit nicht authentifizierten Anfragen.

Ein generischer Ansatz ist die Verwendung der *IP*-Adresse des Anfragestellers. Vorteilhaft ist die von der Applikation entkoppelte Einsetzbarkeit. Allerdings ist das Merkmal nur begrenzt nützlich: Angreifer können z. B. Botnetze oder Cloud-Computing nutzen, um sich einen großen Pool an *IP*-Adressen zu verschaffen. Insbesondere bei Unterstützung von IPv6-Adressen mit 128 *Bit* Länge ist allein das serverseitige Zwischenspeichern der *IP*-Adressen im *RAM* ein potenzielles Ziel für *DDOS*-Attacken. Zudem ist die Metrik unscharf, da sich durch Einsatz von *Network Address Translation (NAT)* mitunter mehrere legitime Nutzer eine öffentliche *IP*-Adresse teilen.
Anstatt der Request-*IP*-Adressen können auch die Accounts betrachtet werden, zu denen Authentifizierungsversuche erfolgen. Ein Rate-Limiting je Account kann zur Folge haben, dass legitime Nutzer sich nicht anmelden können, wenn ein Angriff gegen ihren Account läuft. Dagegen sind beide Ansätze kombinierbar, sodass der Server zum Rate-Limiting Tupel aus *IP*-Adresse und Nutzerkennung vorhält. Statt einzelne Accounts gezielt zu attackieren, kann ein Angreifer dennoch typische Passwörter für viele verschiedene Accounts ausprobieren. Diese Angriffsform ist mitunter nur über ein erhöhtes Anfragevolumen feststellbar. In diesem Fall sind sämtliche Anfragen als potenzielle Angriffe zu werten.

Zur Durchsetzung eines Rate-Limitings je Anfragesteller sind nach [Goo21] folgende Algorithmen einzeln oder in Kombination anwendbar:

- **Fixed-Window**: Ein Limit wird für ein abgestecktes Intervall definiert, z. B. 1 000 Anfragen pro Stunde. An der Grenze zwischen zwei Intervallen beträgt die geduldete Anfragezahl damit das Doppelte des Limits.
- **Sliding-Window**: Das Zeitfenster wird ausgehend von der ersten betreffenden Anfrage aufgespannt. Dadurch werden Anfrage-Spitzen abgefangen.
- **Token-Bucket**: Der Anfragesteller entnimmt je Request einen Token aus einem Eimer, respektive Pool, mit begrenzter Kapazität. Ist der Pool erschöpft, greift das Rate-Limit. Das Kontingent wird langsam aufgefüllt, bis der Eimer wieder voll ist. Dadurch werden Anfrage-Spitzen auch unabhängig von einer Zuordnung zu einem Zeitfenster abgefangen.
- **Leaky-Bucket**: Anders als beim Token-Bucket wird die Ausflussmenge an Tokens aus dem Eimer je Zeiteinheit begrenzt. Durch die Tokens werden die Anfragen in eine Queue eingeordnet. Aufgrund des begrenzten Kontingents kann ein Anfragesteller mit seinen Requests keinen Stau verursachen.

Reaktionsmöglichkeiten bei Überschreitung eines Rate-Limits

Wurde ein Limit überschritten, so hat der Server mehrere, teilweise kombinierbaren Möglichkeiten auf die entsprechende und folgende Anfragen zu reagieren.

Blockierung Der fehlersicherste und standardkonforme Umgang besteht im Abweisen einer durchs Rate-Limiting blockierten Anfrage, z. B. über eine Antwort mit *HTTP*-Statuscode 429: „Too Many Requests“.

Unvorhersagbares Verhalten Da eine simple Blockierung für Angreifer feststellbar ist, werden sie ihr Verhalten anpassen und Wege zur Umgehung suchen. Deshalb kann eine Gegenmaßnahme die Verschleierung des angewandten Rate-Limits sein. Um die Tools von *Script-Kiddies* zu verwirren, kann die Antwort auch variiert werden. Beispielsweise kann bei angewandtem Rate-Limit die Rückgabe einer Erfolgsmeldung zum ungültigen Login-Versuch dazu führen, dass das Angreifer-Tool die Attacke abbricht. Diese Maßnahme ist allerdings nur gegen technisch nicht-versierte Angreifer anzubringen.

Device-Cookies Für bekannte Geräte kann eine Umgehung des Rate-Limitings nach dem Verfahren aus Kapitel 4.2.3 gestattet werden.

Captchas Um die horizontale Skalierung automatisierter Anfragen zu hemmen, wird ein Nachweis über die Menschlichkeit des Anfragestellers eingefordert. Nur wenn der Nachweis vorliegt, darf das betreffende Rate-Limit umgangen werden. Dazu muss der User Rätsel lösen, welche nur durch Menschen und möglichst nicht durch maschinelles Lernen erfüllbar sein sollen. Aus dem Aufwand zum Bearbeiten der Rätsel ergibt sich das *UX*-Problem des Verfahrens.

Client Puzzle Protocols (CPPs) *CPPs* sind eine *UX*-freundliche Alternative zu Captchas. Die Implikation des Client-Hashings, eine Lastverschiebung zum Angreifer, wird abstrahiert und gezielt herbeigeführt. Der Client muss einen gewissen, geleisteten Rechenaufwand nachweisen, bevor er seine Anfrage stellen darf. Dieses Konzept wird als *Proof of Work (PoW)* bezeichnet.
Der Server gibt *CPP*-Nonces an anfragende Clients aus. Ein Client muss dann z. B. aus aktuellem Zeitstempel, Domainnamen und *CPP*-Nonce einen Hashwert bilden, der mit einer bestimmten Bitanzahl an Nullen beginnt. Dazu hängt der Client per Brute-Force selbstgewählte Opaques an die vorgegebenen Werte an, bis ein valider Hashwert erreicht ist. Daraufhin reicht der Client seine Werte beim Server ein, welcher mit nur einer Hash-Operation prüfen kann, ob die Antwort korrekt ist. Zur Steigerung der Clientlast bei gleicher Serverlast können dem Client mehrere Sub-Puzzles auferlegt werden, von denen der Server nur ein zufällig gewähltes auf seine korrekte Beantwortung hin auswertet.[Bos21]

Guided Tour Puzzles Anstatt die *PoW*-Last wie bei *CPPs* an die Rechenkapazitäten der Clients zu binden, wird sie an den Netzwerkdurchsatz gebunden. Der Client muss eine gewisse Anzahl an Servern ansprechen, wobei der angesprochene Server vorgibt, welcher Server als Nächstes anzusprechen ist. Dabei geben die Server den besuchenden Clients jeweils einen clientgestützten Bearer-Token mit. Für Zugriff auf den eigentlichen Applikationsserver ist ein Token vorzulegen, der hinreichend viele Server-Ansprachen nachweist. Dadurch wird einem Angreifer mit hohen Bandbreite-Kapazitäten entgegengewirkt.

5.2 Tests

Zur praktischen Validierung werden Konzepte aus Kapitel 3 und 4 bezüglich ihrer Einsatzfähigkeit für Webapplikationen geprüft. Messergebnisse sowie die Software zur Reproduktion der Tests sind in Anhang D (Digitaler Anhang) zu finden.

5.2.1 Benchmarking zur Hashing-Performance von Webbrowsern

Um das Client-Hashing aus Kapitel 3 im Web umsetzen zu können, muss der Webbrowser die Hash-Funktionen ausführen. Dies kann nativ über die in Kapitel 5.1.2 vorgestellte Web-Crypto-*API* oder anhand einer separaten Bibliothek erfolgen. Für den Test werden folgende Hash-Funktionen gewählt:

- ***PBKDF2***: Wie in Kapitel 2.2.3 dargelegt, ist dieser Algorithmus rechenintensiv, aber nicht speicherhart. Deshalb eignet er sich nicht für neue Protokoll-Designs. *PBKDF2* wurde aufgenommen, da es als einzige langsame Hash-Funktion durch die Web-Crypto-*API* bereitgestellt wird. [Wat17]
- **Bcrypt**: Obwohl Bcrypt bereits 1999 veröffentlicht wurde, findet der Algorithmus aufgrund seiner bewährten Praxistauglichkeit auch heutzutage noch weite Verbreitung. Die Rechenintensität ist als Work-Factor einstellbar. Die Speicherhärte hat einen konstanten Bedarf von $4\,KiB$. Die maximale Passwortlänge ist effektiv auf 71 Zeichen beschränkt.[Whi21]
- **Argon2id**: Argon2 ging 2015 als Gewinner aus der *Password Hashing Competition (PHC)* hervor.[BDK17] In der *PHC* wurde die Angriffsresistenz von Passwort-Hashing-Verfahren evaluiert. In der Modifikation Argon2id ist der Algorithmus einstellbar rechenintensiv sowie speicherhart gegen diverse Angriffe. Der Empfehlung der *PHC* sowie [BDKJ21] folgend, wird der Einsatz von Argon2 in der Variante Argon2id präferiert.

Zur Durchführung des Tests sind konkrete Vergleichswerte für die Kostenparameter zu wählen. Als Referenz dienen die Ergebnisse aus [BDKJ21, Whi21]:

- *PBKDF2* mit *SHA*-256: 310 000 Iterationen
- Bcrypt: Work-Factor von 12
- Argon2id: Speicherbedarf m, Parallelisierung p , Zeitbedarf t
  - Variante 1: $m = 2\,GiB, p = 4\,Lanes, t = 1\,Iteration$
  - Variante 2: $m = 64\,MiB, p = 4\,Lanes, t = 3\,Iterations$

Da das Allozieren von $2\,GiB\ RAM$ in Webbrowsern fehlschlug, wird für Argon2 die zweite Variante verwendet. Als Testdaten kommen jeweils zufällig generierte, achtstellige, alphanumerische Passwörter zum Einsatz.

Als Mitigation gegen Angriffe wie Spectre und Meltdown verfügen die Zeitgeber der Browser-Sandboxes mitunter nur über eine eingeschränkte Genauigkeit im Millisekundenbereich.[GPTY18] Deshalb wird die durchschnittliche Ausführungsdauer des Hashings über 20 Iterationen gemessen. Bei den vorliegenden Algorithmen führt dies zu einer Gesamtberechnungszeit von mindestens einer Sekunde, was für die gewünschte Genauigkeit der Betrachtung ausreicht. Die ermittelte durchschnittliche Ausführungsdauer unterliegt Verzerrungen, z. B. durch *Garbage-Collection* oder Last von Hintergrundprozessen. Zur Stabilisierung wird der Vorgang mit einer experimentell bestimmten Anzahl von fünf Durchläufen wiederholt. Gewertet wird nur der schnellste Durchlauf, da dieser am besten geeignet ist, die Rechengeschwindigkeit der Maschine zu repräsentieren.[2]

Als Referenz zur Beurteilung der Ausführungsgeschwindigkeiten mit Web-Technologien dient die Softwareumgebung Python. Wie in Kapitel 5.1.2 erläutert, wären reine Python-Programme zu langsam und ein Vergleich damit nicht sinnvoll. Jedoch erlaubt Python u. a. über die Schnittstelle *C Foreign Function Interface (CFFI)* das Einbinden vorkompilierter Bibliotheken. Diese können damit performanter agieren als der aufrufende Python-Code. Als umfangreiche Kryptografie-Bibliotheken sind hierzu OpenSSL und Sodium (ehemals NaCl) zu nennen.
Im Webbrowser-Umfeld erlaubt *JavaScript* als interpretierte Sprache nur eine begrenzt performante Ausführung. Deshalb können rechenintensive Operationen als kompilierter Binärcode in *WebAssembly (WASM)* umgesetzt werden. Durch Portierung sind mitunter auch bestehende Bibliotheken via *WASM* ausführbar, z. B. Sodium. Das Konzept der Web-Worker erlaubt es einer Webseite, rechenintensive Operationen vom Hauptthread entkoppelt als Hintergrundthreads auszuführen. Dadurch muss auch ein Passwort-Hashing mit hohen Zeitansprüchen nicht die Web-*UI* blockieren, respektive die *UX* beeinträchtigen.

[2] Zur Verifikation der Methodik wurden die Messungen mit dem Framework „benchmark.js“ (zu finden unter `https://benchmarkjs.com/`) reproduziert.

Tabelle 5.3 stellt die Testergebnisse dar. Die Web-Crypto-*API*, vorgestellt in Kapitel 5.1.2, zeigt eine gute Leistung. Dies ist erwartbar, da sie einer nativen Ausführung entspricht. Währenddessen erzielen die *JavaScript*-Bibliotheken überwiegend schlechte Ergebnisse. Eine Wartezeit von über einer Sekunde wird aus *UX*-Perspektive als untragbar angenommen. Bemerkenswert ist die Leistungsfähigkeit von bcryptjs trotz *JavaScript*-Basis. Die webbasierten Argon2-Pakete führen zu einem schlechten Resultat, obwohl sie auf *WASM* aufsetzen. Im Gegensatz zu Bcrypt benötigt ihre Ausführung 64 MiB anstatt nur 4 KiB RAM. Daher wurde das Laufzeitverhalten in Abhängigkeit vom Speicherbedarf m untersucht, siehe Tabelle 5.4. Hieraus ergibt sich, dass in den durch die getesteten Browser-Engines aufgespannten Sandboxes kurzzeitige Allokationen großer Speichermengen nicht effizient arbeiten.
Für ein performantes, zukunftsweisendes Client-Hashing von Passwörtern im Webbrowser ist daher die Aufnahme einer einstellbar rechenintensiven und speicherharten Hashfunktion, wie Argon2id, in die Web-Crypto-API erstrebenswert.

Tabelle 5.3: Benchmarking zur Hashing-Leistung von Webbrowsern gegenüber nativer Ausführung
Farblich hervorgehoben sind die Werte zur Faktorbestimmung je Verfahren

Verfahren	Bibliothek	Version	Technologie	Plattform[3]	Dauer (in ms)	Faktor
Ausführung auf dem nativen Betriebssystem						
PBKDF2	hashlib	(Python)	OpenSSL	Python 3.8.10	76	1,2
	`cryptography`[1]	3.4.8	OpenSSL		65	1,0
Bcrypt	`bcrypt`[1]	3.1.7	CFFI		201	1,0
Argon2	`pynacl`[1]	1.3.0	CFFI, Sodium		84	1,0
	`argon2-cffi`[1]	18.3.0	CFFI		87	1,0
Ausführung durch die Sandbox des Webbrowsers						
PBKDF2	Web-Crypto-API	(Browser)	Nativ	Firefox[4]	56	0,9
				Chromium[4]	117	1,8
	`node-forge`[2]	0.10.0	JavaScript	Firefox[4]	7577	117
				Chromium[4]	2884	44
	`sjcl`[2,5]	1.0.8	JavaScript	Firefox[4]	1692	26
				Chromium[4]	887	14
Bcrypt	`bcryptjs`[2]	2.4.3	JavaScript	Firefox[4]	398	2,0
				Chromium[4]	317	1,6
Argon2	`argon2-browser`[2]	1.18.0	WASM	Firefox[4]	3863	46
				Chromium[4]	1358	16
	`sodium-plus`[2]	0.9.0	WASM, Sodium	Firefox[4]	4654	55
				Chromium[4]	1356	16

[1] Bibliothek mit angegebenem Namen zu finden unter `https://pypi.org`.
[2] Bibliothek mit angegebenem Namen zu finden unter `https://www.npmjs.com`.
[3] Ausführung der Tests unter Ubuntu Linux 20.04.3 LTS, Kernel 5.11.0-27-generic.
[4] Eingesetzte Webbrowser: Mozilla Firefox 91.0.2, Chromium Browser 93.0.4577.63.
[5] `sjcl` bezeichnet die *Stanford Javascript Crypto Library (SJCL)*.

Tabelle 5.4: Benchmarking zur Hashing-Leistung von Webbrowsern in Abhängigkeit vom Speicherbedarf für Argon2

Speicherbedarf für Argon2		64 MiB	16 MiB	4 MiB	1 MiB
Firefox	`argon2-browser`	3 849 ms	1 004 ms	244 ms	99 ms
	`sodium-plus`	4 639 ms	1 166 ms	319 ms	75 ms
Chromium	`argon2-browser`	1 373 ms	377 ms	126 ms	34 ms
	`sodium-plus`	1 391 ms	379 ms	132 ms	35 ms

Messungen im gleichen Setup wie zuvor dargestellt, siehe Tabelle 5.3.
Ausführungszeiten gemessen nach dem oben erläuterten Verfahren.

5.2.2 Vergleich der Hashing-Leistung verschiedener Endgeräte

Die bisherige Test-Betrachtung bezieht sich auf nur ein Endgerät, das für Consumer-Hardware durchaus als leistungsstark bezeichnet werden kann.[3] Ein Client-Hashing muss jedoch potenziell auf diversen Endgeräten ausführbar sein. Daher gilt es, auch leistungsschwächere sowie mobile Endgeräte zu betrachten. Tabelle 5.5 listet relevante Merkmale der für den Test ausgewählten Geräte auf.

Tabelle 5.5: Ausgewählte Endgeräte zum Testen der Hashing-Leistung

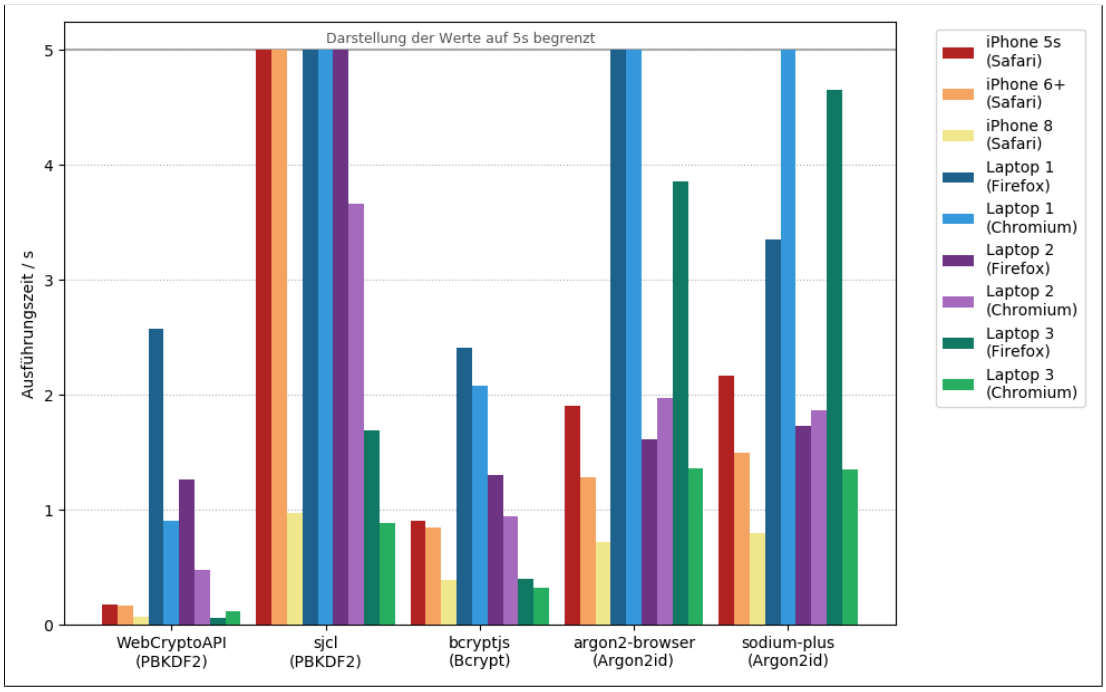
Gerät	**CPU**	**RAM**	**Betriebssystem**	**Webbrowser**
iPhone 5s	Apple A7 2×1,3 GHz (ARMv8)	1 GiB	iOS 12.5.4	Safari 12.1.2
iPhone 6 Plus	Apple A8 2×1,4 GHz (ARMv8)	1 GiB	iOS 12.5.4	Safari 12.1.2
iPhone 8	Apple A11 4×1,6 GHz+ 2×2,39 GHz (ARMv8)	2 GiB	iOS 14.7.1	Safari 14.1.2
Laptop 1	AMD A4-1200 APU 2×1,0 GHz (x86-64)	1,5 GiB	Ubuntu Linux 20.04.3 LTS	Firefox und Chromium entsprechend Tabelle 5.3
Laptop 2	Intel Pentium N3700 4×1,6 GHz (x86-64)	4 GiB		
Laptop 3	AMD Ryzen 5 4600H 12×3 GHz (x86-64)	16 GiB		

Der Test wird entsprechend der Beschreibung aus Kapitel 5.2.1 ausgeführt, wobei auch die in Tabelle 5.3 genannten Versionen an Programmen und Bibliotheken zum Einsatz kommen.[4] Abbildung 5.7 stellt die Testergebnisse dar. Wie erwartet, fällt die Ausführungszeit bei leistungsschwacher Hardware um ein Vielfaches höher aus. Wie Abbildung 5.7b entnehmbar, bleibt in einer Python-Umgebung

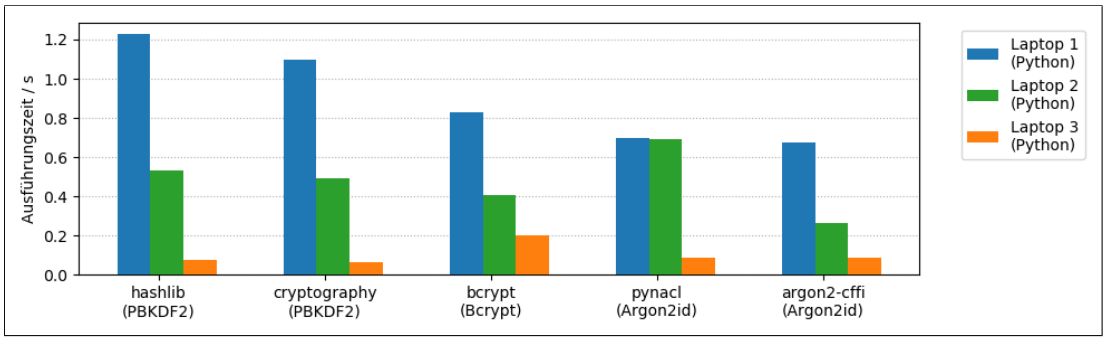
[3] Zur Durchführung des Tests aus Kapitel 5.2.1 wurde das in Tabelle 5.5 als „Laptop 3“ aufgeführte Gerät verwendet.

[4] Einzig das Paket `node-forge` wird aufgrund des besonders schlechten Laufzeitverhaltens nicht erneut betrachtet.

die Nutzung der Verfahren dennoch praktikabel. Für Webanwendungen gilt dies nur eingeschränkt, siehe Abbildung 5.7a. Bemerkenswert ist, dass die Apple-Geräte für die Web-Crypto-*API*, Bcrypt sowie Argon2 trotz beschränkter Leistungsmerkmale besonders gute Ergebnisse erzielen, siehe Abbildung 5.7a. Dies ist durch Optimierungen an Hardware sowie Browser-Software erklärbar und verdeutlicht, dass die Nutzbarkeit von Client-Hashing nicht nur an die Kennzahlen der Hardware-Ressourcen, sondern auch die Güte der Softwareunterstützung gebunden ist. Diese experimentell zu bestimmenden Faktoren gilt es bei Einführung einer Applikation mit Client-Hashing zu berücksichtigen.



(a) Plattform Webbrowser



(b) Plattform Python

Abbildung 5.7: Vergleich der Hashing-Leistung verschiedener Endgeräte
Quelle: Eigene Darstellung

Um Client-Hashing nutzbar zu machen, ist eine obere Schwelle für die Ausführungszeiten zu unterstützender Endgeräte festzulegen. Für das Beispiel aus Abbildung 5.7a ergibt sich bei einer maximal tolerierten Wartezeit von drei Sekunden

die Möglichkeit zum Einsatz der Web-Crypto-*API* oder bryptjs. Diese Zeitmessungen sind jedoch auf ein sich sonst im Leerlauf befindliches System bezogen, sodass die Wartezeiten bei Last durch andere Nutzerprozesse auch deutlich höher ausfallen könnten. Problematisch ist zudem, dass Nutzer leistungsschwacher Endgeräte systematisch benachteiligt werden. Dies kann für den Anbieter der Applikation u. a. zu einem Wettbewerbsnachteil oder Kundenbeschwerden führen. Es birgt weiterhin die Gefahr, Menschen an der digitalen Teilhabe zu hindern oder auszuschließen. Da die Kostenvorgaben zum Passwort-Hashing an die aktuellen technischen Entwicklungen angepasst werden, ist auch nicht zu erwarten, dass dieses Ungleichgewicht zwischen Geräten zukünftig schwindet.
Daher muss vor dem Einsatz von Client-Hashing die Zielgruppe möglicher Nutzer klar umrissen werden. Im Zweifelsfall ist von einem Client-Hashing abzusehen oder abgeschwächte Formen wie das dynamische Server-Relief-Hashing in Kombination mit *JIT*-Hashing anzuwenden, siehe Kapitel 3.1.1 und 3.2.2.

5.2.3 Benchmarking zum Request-Durchsatz des Servers

Clientseitige Passwortverarbeitung ändert nicht nur das Laufzeitverhalten der Clients, sondern auch das der Server. Um die Auswirkungen beurteilen zu können, wird für diesen Test die durch parallele Anmeldevorgänge entstehende Serverlast betrachtet. Für die Untersuchung sind folgende Betriebsmodi zu unterscheiden:

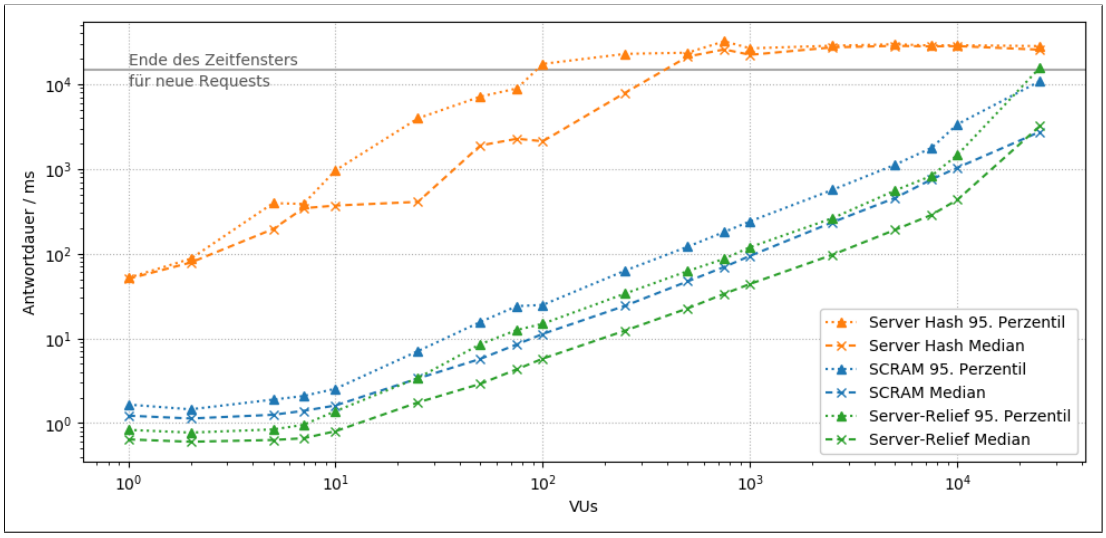
- Ausschließliches **Server-Hashing**, entsprechend Kapitel 2.3.1.
- Aufwandsarmes **Server-Relief-Hashing** durch Ergänzung eines ressourcenintensiven Client-Hashings, entsprechend Kapitel 3.2.2.
- Server-Relief via **SCRAM**, entsprechend Kapitel 3.3.3.

Folgende Algorithmen werden zum serverseitigen Testen der Modi gewählt:

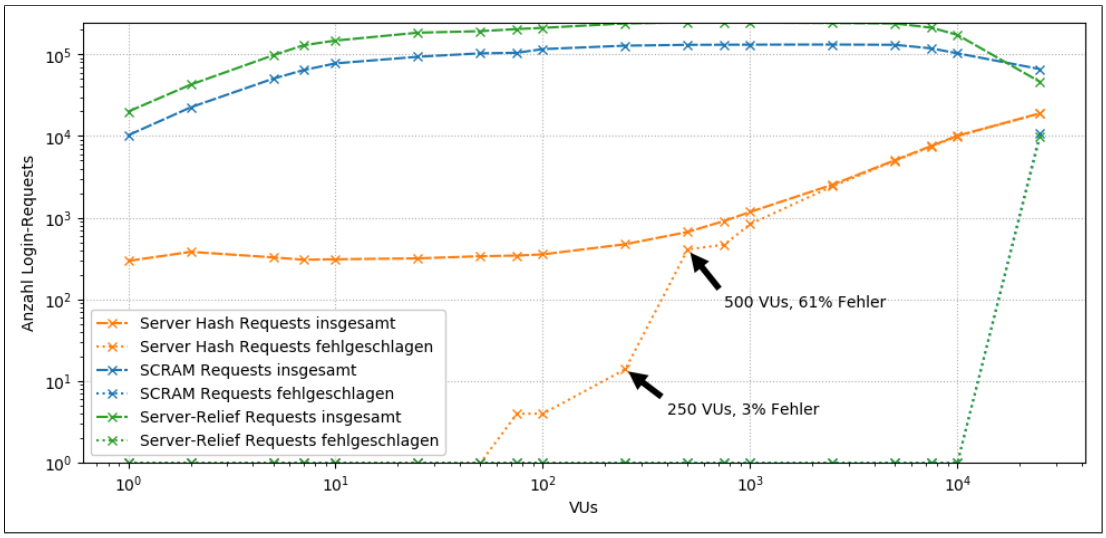
- **Server-Hashing**: Argon2id, entsprechend Kapitel 5.2.1.
- **Server-Relief-Hashing**: *SHA*-256.
- **SCRAM**: *HMAC* mit *SHA*-256.

Die Simulation paralleler Nutzer erlaubt eine Einschätzung der Belastbarkeit des Systems[5] in Abhängigkeit des durch die Nutzerzugriffe erzeugten Ressourcenverbrauchs. Dazu wird eine ansteigende Anzahl automatisierter Anfragesteller, genannt *Virtual Users (VUs)*, eingesetzt. Jeder *VU* sendet bis zur Überschreitung der Dauer eines festgelegten Zeitfensters sequenziell so viele Requests wie möglich. Als Zeitfenstergröße wurden jeweils 15 Sekunden gewählt, um den Einfluss von Verfälschungen durch kurzzeitige Lastschwankungen zu minimieren und

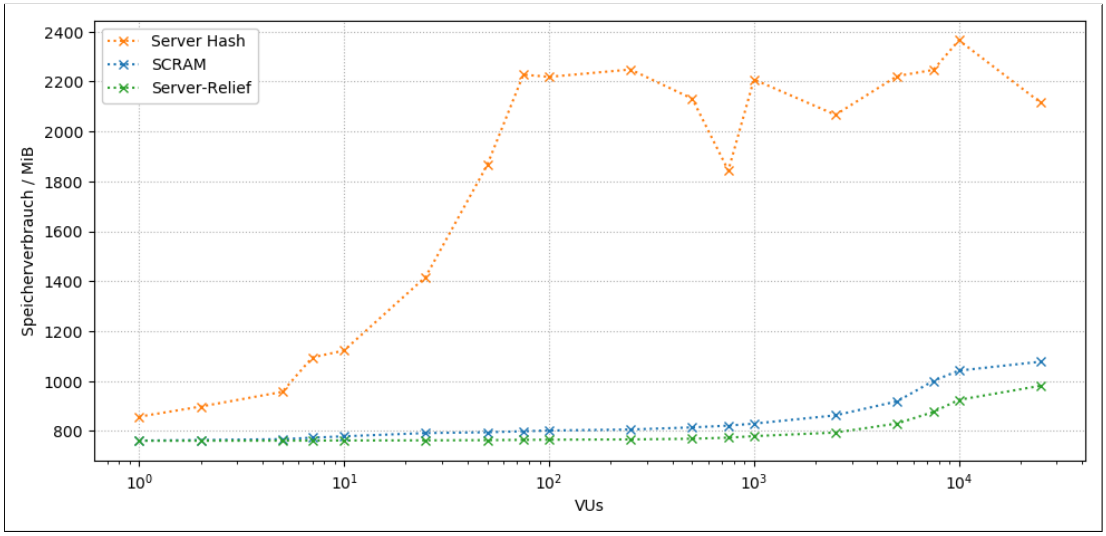
[5] Als Hardware wird für den Test das in Tabelle 5.5 als „Laptop 3“ bezeichnete Gerät eingesetzt.



(a) Dauer zur Beantwortung von Login-Requests



(b) Anzahl und Anteil beantworteter Login-Requests



(c) Ressourcenverbrauch zur Beantwortung von Login-Requests

Abbildung 5.8: Benchmarking zum Request-Durchsatz des Servers
Quelle: Eigene Darstellung

damit stabilere Messergebnisse zu erhalten.[6] Abbildung 5.8 zeigt die Ergebnisse des Benchmarking-Tests. Wie erwartet, benötigt Server-Hashing deutlich mehr Rechenzeit, sodass die Antwortdauer höher ausfällt, siehe Abbildung 5.8a. Damit erreicht auch der Request-Durchsatz ein schlechteres Ergebnis und die Skalierbarkeit ist sehr begrenzt: Bereits bei 500 *VUs* schlagen mehr als die Hälfte der Requests fehl, siehe Abbildung 5.8b. Der *RAM*-Verbrauch steigt durch das Server-Hashing kontinuierlich, solange der Server die Anfragen noch beantworten kann und stagniert danach entsprechend, siehe Abbildung 5.8c. Der hohe, initiale *RAM*-Verbrauch liegt darin begründet, dass die Server-Software zur Performance-Steigerung mit 24 geforkten Arbeiterprozessen startet, was der doppelten *CPU*-Kernanzahl[5] entspricht.
Server-Relief-Hashing und *SCRAM* zeigen bei Skalierung einen deutlich besseren Request-Durchsatz. Die Werte für *SCRAM* fallen stets geringfügig schlechter aus. Dies liegt einerseits darin begründet, dass *SCRAM* zwei *Roundtrips* für eine Authentifizierung benötigt, siehe Abbildung 5.8a und 5.8b. Außerdem ist der *RAM*-Verbrauch durch das temporäre Vorhalten der Challenges leicht erhöht, siehe Abbildung 5.8c. Die Messungen bestätigen die vorangegangene, theoretische Betrachtung: Server-Relief-Hashing ist im Gegensatz zu Server-Hashing performant skalierbar. Der Overhead durch den Einsatz von *SCRAM* ist vertretbar.

5.2.4 Einsatzfähigkeit von Security-Tokens mittels FIDO2

In Kapitel 4.3.3 wurde das Konzept der Security-Tokens in Kombination mit dem FIDO2-Standard vorgestellt. Es gilt zu evaluieren, inwiefern Security-Tokens zur Web-Authentifizierung bereits heutzutage einsatzfähig und nutzbar sind. Dazu werden exemplarisch die in Tabelle 5.6 aufgeführten Geräte betrachtet.

Tabelle 5.6: Zum Test der Einsatzfähigkeit evaluierte Security-Tokens

Hersteller	Yubico	Nitrokey
Modell	Security Key NFC by Yubico	Nitrokey FIDO2
Protokolle	FIDO2, FIDO U2F	FIDO2, FIDO U2F
Transporte	NFC, USB-A	USB-A
UP-Nachweis	Via Touch-Button	Via Touch-Button
Resident-Keys	25 Keys	50 Keys

Tabelle 5.7 stellt die Testergebnisse dar. Die Mehrheit der Browser bietet eine grundlegende Unterstützung. Eine Ausnahme bildet der TOR Browser, bei

[6] Zur Durchführung der Messungen wurde das Framework „k6“ (zu finden unter `https://k6.io/`) in einer *Docker*-Umgebung eingesetzt, siehe Anhang D (Digitaler Anhang).

dem *WebAuthn* absichtlich deaktiviert ist, was vermutlich zur Verringerung der Angriffsoberfläche dient. Wie den Anmerkungen zu Tabelle 5.7 entnommen werden kann, variiert trotz FIDO2-Unterstützung die Implementierungsgüte teilweise stark und sicherheitsrelevante Features fehlen mitunter. So sollte immer eine Nutzerbestätigung via *UP* oder *UV* eingefordert werden. Wenn eine Funkschnittstelle wie *NFC* als Transport genutzt wird, ist das Einfordern einer *UV* (anstatt *UP*) zur FIDO2-Anmeldung obligatorisch. Andernfalls könnte ein Angreifer mit seinem Smartphone im Vorbeigehen eine unbemerkte Kommunikation mit dem Authenticator des Opfers vornehmen.
Eine grundlegende Unterstützung des FIDO2-Standards ist gegeben. Um die Technik plattformübergreifend stabil verwenden zu können, muss die Softwareunterstützung ausgebaut werden.

Tabelle 5.7: Einsatzfähigkeit von Security-Tokens via FIDO2 für ausgewählte Webbrowser auf unterschiedlichen Betriebssystemen

Betriebssystem	**Webbrowser**			**Security-Tokens**	
	Name	**Basis**[1]	**WebAuthn**	**Yubico**	**Nitrokey**
Ubuntu Linux 20.04.3 LTS	Mozilla Firefox 92.0	Gecko Engine	(✓)[2,3]	✓	(✓)[2,4]
Ubuntu Linux 20.04.3 LTS	TOR Browser 10.5.6	Mozilla Firefox	×	×	×
Ubuntu Linux 20.04.3 LTS	Chromium 93.0.4577.63	Blink/V8 Engine	✓[5]	✓	(✓)[2,4]
Ubuntu Linux 20.04.3 LTS	Opera 78.0.4093.184	Chromium	✓[5]	✓	(✓)[2,4]
Ubuntu Linux 20.04.3 LTS	Gnome Web/ Epiphany 3.36.4	Webkit Engine	×	×	×
Apple iOS 14.7.1	Safari 14.1.2	Webkit Engine	✓[5]	✓	×[6]
Android 10 FP3	Google Chrome 93.0.4577.62	Blink/V8 Engine	(✓)[7]	✓	×[6]
Android 10 FP3	Mozilla Firefox Mobile 92.1.1	Gecko Engine	(✓)[7]	✓	×[6]
Microsoft Windows 10 2004	Mozilla Firefox 92.0	Gecko Engine	✓[5]	✓	(✓)[2,4]
Microsoft Windows 10 2004	Microsoft Edge 92.0.902.78	Chromium	✓[5]	✓	(✓)[2,4]
Microsoft Windows 10 2004	Internet Explorer 11	Trident Engine	×	×	×

[1] Die technologische Grundlage, auf welcher der Webbrowser aufbaut.
[2] Keine Unterstützung einer Nutzerbestätigung via *UV*.
[3] *WebAuthn*-Parameter `timeout` zur Steuerung der eingeräumten Zeitspanne zur Erbringung des *UP*-Nachweises wird ignoriert.
[4] Keine Unterstützung von Resident-Keys.
[5] Nachweis wird mittels *UV* ausgeführt, auch wenn explizit eine *UP* angefordert wurde.
[6] Technisch auf dem Smartphones nicht möglich, da nur *NFC* als Transport vorhanden.
[7] Keine Unterstützung einer Nutzerbestätigung via *UP* oder *UV*.

5.3 Bewertung

Zu der Realisierung wurden anhand des entwickelten Prototyps Aspekte der Implementierung betrachtet und mittels Tests die Möglichkeiten einer praktischen Umsetzung aufgezeigt. Abschließend gilt es, die Realisierung der Verfahren zur clientseitigen Passwortverarbeitung sowie optionalen *MFA* aus Sicherheitsperspektive bezüglich der Programmierung zu bewerten. Hierzu erfolgt nacheinander eine prototypische Implementierung der Applikation in folgenden drei Varianten entsprechend der Vorgaben aus Kapitel 5.1.1:

- Variante 1: *1FA* mit Server-Hashing
- Variante 2: *1FA* mit *SCRAM*
- Variante 3: *MFA* mit *SCRAM*, *TOTP* und FIDO2

Als Metrik für die Bewertung dient die Softwarekomplexität, wobei nur die für die Verfahren zwingend notwendige Komplexität und nicht die tatsächliche des Prototyps relevant ist. Jedes Feature besitzt inhärent eine individuelle Grundkomplexität, sodass die Gesamtkomplexität mit zunehmender Feature-Anzahl steigt. Jedoch erlaubt eine saubere Code-Strukturierung, die Komplexität handhabbar zu machen und Synergien zwischen Features zu nutzen, sodass die Gesamtkomplexität niedriger als die Summe der Einzelkomplexitäten ausfallen sollte. Des Weiteren ist Komplexität kapselbar, z. B. in Bibliotheken, sodass zwischen Anwendungskomplexität und Verfahrenskomplexität zu unterscheiden ist. Beispielsweise erzeugt die Implementierung und Optimierung eines Hashing-Verfahrens nennenswerte Komplexität, während der Aufruf dieser Funktion durch eine Applikation trivial zu bewerkstelligen ist. Mit zunehmender, nicht-kapselbarer Komplexität der Implementierung sinkt die Übersichtlichkeit und Verständlichkeit des Quellcodes. Damit steigt die Wahrscheinlichkeit für Programmierfehler und die Wartbarkeit sowie Erweiterbarkeit bestehender Features werden beeinträchtigt. Zur Messung einzelner Ausprägungen von Komplexität wurden Verfahren entwickelt, wie z. B. die *Zeilenmetrik*, die allerdings jeweils nur eine begrenzte Aussagekraft besitzen. Beispielsweise kann eine hohe Anzahl an Quellcode-Zeilen Folge einer hohen Komplexität sein. Die hohe Anzahl könnte aber auch durch einen sehr expliziten und damit aussagekräftigen Quellcode entstehen. Deshalb wird im Folgenden die während der Implementierung wahrgenommene Komplexität thematisiert. Die prototypische Anwendung besitzt durch Management von Accounts und Sessions als Baseline eine Grundkomplexität. Es gilt die zusätzliche Komplexität durch die drei oben genannten Varianten zu betrachten.

Variante 1 besitzt durch das ausschließliche Server-Hashing die geringste Komplexität der drei Varianten und ist stringent umsetzbar. Sowohl die Überprüfung

zur Passwortstärke als auch die Aktualisierung der Hashing-Parameter lassen sich serverseitig intern steuern, siehe Kapitel 2.2.3.
Variante 2 erfordert kryptografisches Hashing durch Client und Server, was die Komplexität erhöht. Zusätzlich müssen auch die Hashing-Parameter sowie die Überprüfung zur Passwortstärke über die *API* vom Server an die Clients exponiert werden. *API*-Umfang und Client-Server-Kommunikation nehmen zu, sodass die Angriffsoberfläche wächst. Jedoch bedarf die Umstellung vom Server-Hashing zum Server-Relief-Hashing nur absehbare Änderungen an Authentifizierungsfunktionen mit lokal begrenzten Auswirkungen. Die Steigerung der Komplexität ist damit überschaubar sowie kontrollierbar.

Zu Variante 3 gilt es, neben der iterativen *MFA* selbst auch die Verfahren für die *MFA*-Faktoren zu betrachten:
TOTP besitzt eine äußerst geringe Verfahrenskomplexität, sodass die Implementierung trivial erscheint. Jedoch darf die Anwendungskomplexität nicht vernachlässigt werden. Als Constraints müssen durch die Anwendung u. a. ein sinnvolles Zeitfenster forciert sowie eine Wiederverwendung von *TOTPs* verhindert werden, siehe Kapitel 4.2.2. In einem unsachgemäßen Umgang mit dieser Komplexität besteht die Hauptfehleranfälligkeit des Verfahrens.
Im Gegensatz zum niederschwelligen *TOTP*-Verfahren besitzt FIDO2 eine inhärent höhere Verfahrenskomplexität. Um FIDO2-Protokollabläufe durchlaufen zu können, muss die Software von Webserver, Webclient, Webbrowser und Authenticator zusammenarbeiten. Die Anwendungskomplexität fällt im Frontend durch Einhaltung der Protokolle *CTAP* und *WebAuthn* hoch, aber überschaubar aus. Im Backend können Kryptografie-Bibliotheken wie OpenSSL zur Kapselung eingesetzt werden, siehe Kapitel 5.2.1. Für FIDO2 ergibt sich eine hohe Verfahrenskomplexität. Aufgrund der diversen Parameter des Verfahrens, wie Transportarten, Nutzerbestätigungen oder Attestierungen, kann eine Kapselung nur begrenzt Komplexität abstrahieren, sodass auch die Anwendungskomplexität durch Einsatz des FIDO2-Verfahrens beträchtlich erhöht wird.
Die iterative *MFA* besitzt kaum eigene Verfahrenskomplexität, da lediglich das Konzept der tokenbasierten Authentifizierung angewendet wird. Jedoch bedarf der Übergang von Variante 2 zu 3 einer massiven Änderung der Geschäftslogik: Es werden zusätzliche Einstellungsoptionen zur Faktorverwaltung für die *MFA* benötigt. Über diese Optionen erfolgen Registrierung, Verifikation und Löschung von *MFA*-Faktoren sowie die Bestimmung des *MFA*-Levels. Neben der Faktor-Registrierung müssen auch für jedes *MFA*-Verfahren die Funktionen zur Prüfung des Faktor-Nachweises implementiert werden. Das Risiko hierbei ist besonders hoch, da die Umgehung des Faktor-Nachweises für ein einzelnes *MFA*-Verfahren

mitunter schon zur Umgehung der *MFA* an sich genügt. Dieser Angriff wird auch als *MFA*-Bypass bezeichnet. Ein Bug in der Umsetzung eines der unterstützten *MFA*-Verfahren senkt daher die Sicherheit des gesamten *MFA*-Prozesses. Wenn die *MFA* verschiedene Verfahren, wie *TOTP* oder FIDO2, unterstützt, steigt aufgrund der Komplexität dieser Verfahren auch zwangsläufig die Fehleranfälligkeit des gesamten Authentifizierungsvorgangs.
Die iterative *MFA* wird nicht nur zum Login, sondern auch für weitere Aktionen innerhalb der Applikation benötigt, siehe Kapitel 4.1.4. Damit müssen über die *API*-Endpunkte zur *MFA* unterschiedliche Server-Handlungen ausgelöst werden, z. B. Anmeldung, Auslösung einer Transaktion oder auch Account-Löschung. Dies erschwert die programminterne Strukturierung, da die Kopplung zwischen Authentifizierung und den geschützten Aktionen erhöht ist, was sich z. B. in der Gestaltung der *MFA*-Tokens entsprechend Kapitel 4.1.2 widerspiegelt. Damit steigert eine *MFA* die Anwendungskomplexität nicht nur durch die zusätzlich benötigten Features, sondern auch durch die geschaffene Abhängigkeit zu anderen Komponenten der Applikation.

Der ursprüngliche Gedanke zum Einfügen der *MFA* in die *IDP*-Applikation zielte auf eine Erhöhung des Sicherheitsniveaus der Authentifizierung ab. Das durch eine Passwort-*1FA* erreichbare Niveau wurde als zu gering eingeschätzt. Aus Perspektive der Komplexitätsbetrachtung ist jedoch eine *1FA* gegenüber einer *MFA* aufgrund der geringeren Komplexität zu bevorzugen. Zur Komplexitätssenkung ist daher eine starke *1FA* anzustreben, z. B. über Security-Tokens. Falls aus Gründen der Abwärtskompatibilität jedoch weiterhin eine *MFA* benötigt wird, lässt sich durch Etablierung einer starken *1FA* allein die geschaffene Komplexität nicht entfernen. Dann gilt es, eine *MFA* mit möglichst wenig verschiedenen Authentifizierungsverfahren umzusetzen.
Eine Implementierung mit möglichst geringer Komplexität ist aus den oben erläuterten Gründen sinnvoll. Zur Erreichung der Komplexitätsreduzierung empfiehlt sich die ausschließliche Nutzung einer starken *1FA*.

6 Fazit

6.1 Zusammenfassung

Die Hardware-Kapazitäten zum Brechen von Passwort-Hashes steigen von Jahr zu Jahr, während die menschliche Lernfähigkeit für komplexe Passwörter konstant bleibt. Ein immer aufwändigeres Passwort-Hashing ist notwendig, was im verteilten System typischerweise das Backend belastet. Da auch die Leistungsfähigkeit der Nutzer-Endgeräte steigt, stellt die Verlagerung des Hashingaufwands zum Frontend einen konsequenten Entwicklungsschritt dar. Diese Thesis diskutiert und beschreibt die zu berücksichtigenden Aspekte zur sicheren Konzeption clientseitiger Passwortverarbeitung. Wie herausgestellt, ist eine sichere Gestaltung im Rahmen des via passwortbasierter Authentifizierung erzielbaren Sicherheitsniveaus möglich und umsetzbar. Da insbesondere im Internet ein höheres Schutzlevel anzustreben ist, betrachtet die Thesis zusätzlich Verfahren zur Mehrfaktorauthentifizierung mit Schwerpunkt auf Besitzmerkmalen. Diese sind zur Stärkung der Sicherungsstufe von Authentifizierungsprozessen geeignet, wobei insbesondere der Einsatz von Security-Tokens empfehlenswert ist. Einen Überblick über die in der Thesis untersuchten Verfahren sowie Risiken bildet Anhang A ab.

6.2 Handlungsempfehlung zu Webapplikationen

Da rein webbasierte Softwarelösungen in absehbarer Zukunft weitere Verbreitung finden, sollte die Webbrowser-Plattform genutzt werden, um generische Anwendungsaspekte aus Webapplikationen zu lösen und über den *UA* bereitzustellen. Die Web-Authentifizierung ist generisch genug, um sie clientseitig als abstrahierendes Framework durch den Webbrowser mit einer *API* zur Webseite zu arrangieren. Dadurch ließen sich Sicherheitsmechanismen zentral organisieren, anstatt diese ähnliche Programmlogik für jede Webapplikation erneut zu implementieren. Allein die hieraus resultierende Angabe, ob ein Nutzer momentan angemeldet ist, würde es Webbrowsern ermöglichen, ihr Verhalten danach auszurichten, z. B. bezüglich Privacy-Maßnahmen. Einen wichtigen Baustein bildet der FIDO2-Standard für Security-Tokens. Jedoch müssten für eine breite Adaptierbarkeit auch Passwörter und *OTPs* Teil dieses Frameworks zur Web-Authentifizierung sein. Wie hervorgehoben, stellt Passwort-Phishing im Web ein gravierendes Problem dar. Durch Implementierung einer Passworteingabe am Webbrowser sowie clientseitiger Passwortverarbeitung durch den Webbrowser wäre hiergegen eine problemlösende Origin-Bindung erzielbar. Diese ließe sich z. B. durch eine abgewandelte Form von *SCRAM* mit Salting durch den Domainnamen realisieren.

6.3 Fortführung der Thesis

Diese Thesis fokussiert Verfahren zur Passwortverarbeitung, welche Hashing als kryptografisches Primitiv nutzen. In einer Weiterführung könnten alternative Ansätze untersucht werden, die asymmetrische Kryptografie einsetzen. Hierzu ist die Protokollfamilie *Password-Authenticated Key Agreements (PAKE)* zu nennen. Der Grundgedanke besteht in einer ergänzenden Authentifizierung zum Diffie-Hellman-Schlüsselaustausch ohne Übertragung von Passwortdetails.
In einer Weiterführung sollten ebenfalls Rate-Limiting-Verfahren inklusive Integration in das *IDP*-Protokoll vertiefend betrachtet werden. Gleiches gilt für die Ausgestaltung zur Einbindung biometrischer Merkmale und Verfahren.
Möglichkeiten der *Risikobasierten Authentifizierung (RBA)* wurden ebenfalls nur gestreift, da sie vom Kernthema wegführen. *RBA* versucht adaptiv aus vergangenen Interaktionen ein Modell für typisches Nutzerverhalten zu entwickeln. Als Datenpunkte lassen sich Tageszeit, Standort, Gerätekennung usw. auswerten. Anhand des Modells wird jeder angeforderten Aktion eine Risikobewertung zugeordnet, die im Zweifelsfall eine Ausführung verhindert.

6.4 Ausblick zur technischen Entwicklung

Es ist anzunehmen, dass eine *1FA* mit Passwortkenntnis zunehmend durch leistungsfähigere Hardware in seiner Sicherheitsfunktion bedroht oder beeinträchtigt wird. Das immer aufwändigere Hashing belastet die Infrastruktur auch mit legitimen Anfragen. Kurzfristig kann durch Etablierung einer *MFA* das Passwort als Faktor mit mangelhaften Sicherheitseigenschaften entschärft werden.
Security-Tokens besitzen das Potenzial, eine sichere Authentifizierung mit Public-Key-Verfahren als *1FA* zu ermöglichen. Mit steigender Verbreitung der Unterstützung für entsprechende Geräte ist zu erwarten, dass Passwörter zukünftig an Bedeutung verlieren werden, bis hin zur ausschließlichen passwortlosen Authentifizierung. Der Einsatz von Passwörtern beschränkt sich dann lediglich auf die lokale Absicherung des Token-Schlüsselmaterials. Mittelfristig wäre also das Passwort als Faktor vollständig durch Security-Tokens ersetzbar.
Langfristig wird Quanten-Computing die Grundannahmen über die Eigenschaften eingesetzter kryptografischer Primitive ändern. Es wird vermutet, dass die Sicherheit asymmetrischer Verfahren stärker beeinträchtigt werden könnte als die symmetrischer Hashing-Verfahren. Falls asymmetrische Kryptografie keine hinreichenden Sicherheitsgarantien mehr bieten sollte, gilt dies ebenso für die vorgestellten, darauf aufbauenden Public-Key-Authentifizierungsmechanismen. Damit könnten Passwörter, die als symmetrische Merkmale davon weniger betroffen sind, dann eine Renaissance erleben und wieder stärker in den Fokus rücken.

7 Thesen

Untersuchung zur Sicherheit von Protokollen mit clientseitiger Passwortverarbeitung und optionaler Mehrfaktorauthentifizierung

- Clientseitige Passwortverarbeitung erhöht gegenüber serverseitiger die Anwendungskomplexität und damit die Fehleranfälligkeit des Systems.
- Clientseitige Passwortverarbeitung verbessert effektiv Skalierbarkeit sowie Ressourcennutzung des Backends zulasten erhöhter Wartezeiten für Nutzer leistungsschwacher Endgeräte. Der Aufwand zum Passwort-Hashing wird umverteilt und nicht vermieden.
- Clientseitiges Passwort-Hashing stellt eine geeignete Methode zur sicheren Gestaltung passwortbasierter Authentifizierungsprozesse mit minimaler Serverlast (*Server-Relief*) dar.
- Salting ist eine obligatorische Maßnahme zum Client-Hashing von Passwörtern. Währenddessen ist ein clientseitiges Peppering zwar möglich, aber nur unter speziellen Voraussetzungen sicher und zielführend anwendbar.
- *Salted Challenge Response Authentication Mechanism (SCRAM)* ist ein solides Framework zur Etablierung clientseitiger Passwort-Verarbeitung mit optionaler, wechselseitiger Authentifizierung.
- Ob Client-Hashing eine Einblicknahme von Passwörtern durch den Server verhindern kann, hängt vom Vertrauensmodell der Software-Architektur ab. Insbesondere für Webapplikationen ist dies nicht der Fall.
- Ein optionaler, iterativer Prozess zur *Mehrfaktor-Authentifizierung (MFA)* kann die systemische Sicherheit der Account-Verwaltung steigern. Hierzu gilt es, eine *MFA* weitläufig für möglichst viele Accounts einzusetzen.
- Passwörter und *One Time Passwords (OTPs)* sind aufgrund mangelnder Origin-Prüfung nicht als ausschließliche Faktoren einer *MFA* für Webapplikationen oder ähnliche Kontexte mit Phishing-Anfälligkeit geeignet.
- Security-Tokens erlauben durch Protokolle wie FIDO2 eine nachhaltig sichere Gestaltung von *MFA*-Vorgängen. Dazu sind die Tokens in Kombination mit Faktoren der Kategorie Wissen oder Inhärenz zu verwenden.

Literaturverzeichnis

[Ala18] ALAM, Masoom: *STRIDE/DREAD Analysis - Threat Modelling of Trinity Wallet*. https://files.iota.org/trinity/Threat+Modelling+Report+V1.2.signed.pdf, 2018. – Zugriff am: 18.07.2021

[AZW10] ALTMAN, Jeffrey E. ; ZHU, Larry ; WILLIAMS, Nicolás: Channel Bindings for TLS / RFC Editor. http://dx.doi.org/10.17487/RFC5929. Internet Requests for Comments, Juli 2010. – RFC 5929. – ISSN 2070–1721

[Bac11] BACHFELD, Daniel: *RSA-Hack könnte Sicherheit von SecurID-Tokens gefährden*. https://www.heise.de/security/meldung/RSA-Hack-koennte-Sicherheit-von-SecurID-Tokens-gefaehrden-1210245.html, März 2011. – Zugriff am: 20.07.2021

[BDK17] BIRYUKOV, Alex ; DINU, Daniel ; KHOVRATOVICH, Dmitry: Argon2: the memory-hard function for password hashing and other applications / University of Luxembourg. März 2017. – Version 1.3 of Argon2: PHC release. – https://raw.githubusercontent.com/P-H-C/phc-winner-argon2/master/argon2-specs.pdf

[BDKJ21] BIRYUKOV, Alex ; DINU, Daniel ; KHOVRATOVICH, Dmitry ; JOSEFSSON, Simon: The memory-hard Argon2 password hash and proof-of-work function / Internet Engineering Task Force. Work in Progress, März 2021 (draft-irtf-cfrg-argon2-13). – Internet-Draft. – https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-argon2-13

[BHJ+21] BRADLEY, John ; HODGES, Jeff ; JONES, Michael B. ; LINDEMANN, Rolf ; VERREPT, Johan: *Client to Authenticator Protocol (CTAP)*. Proposed Standard, Juni 2021. – https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.pdf – Zugriff am: 30.08.2021

[Bos21] BOSTANOV, Vladimir: Client Puzzle Protocols as Countermeasure Against Automated Threats to Web Applications. In: *IEEE Ac-*

cess 9 (2021), S. 75722–75728. http://dx.doi.org/10.1109/ACCESS.2021.3082037. – DOI 10.1109/ACCESS.2021.3082037

[BSI21] BSI: TR-02102-2: Kryptographische Verfahren: Empfehlungen und Schlüssellängen – Teil 2 – Verwendung von Transport Layer Security (TLS) / Bundesamt für Sicherheit in der Informationstechnik. https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/Technische-Richtlinien/TR-nach-Thema-sortiert/tr02102/tr02102_node.html. 2021. – Technische Richtlinie. – Zugriff am: 18.08.2021

[BSP+08] Boeyen, Sharon ; Santesson, Stefan ; Polk, Tim ; Housley, Russ ; Farrell, Stephen ; Cooper, Dave: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile / RFC Editor. http://dx.doi.org/10.17487/RFC5280. Internet Requests for Comments, Mai 2008. – RFC 5280. – ISSN 2070–1721

[CJMS14] Chang, Donghoon ; Jati, Arpan ; Mishra, Sweta ; Sanadhya, Somitra K.: Rig: A simple, secure and flexible design for Password Hashing, Version 2.0. (2014), September. https://www.password-hashing.net/submissions/specs/RIG-v2.pdf. – Zugriff am: 13.08.2021

[Con15] Contini, Scott: *Method to Protect Passwords in Databases for Web Applications.* Cryptology ePrint Archive, Report 2015/387, 2015. – https://eprint.iacr.org/2015/387

[Cro20] Croley, Sam: *What the Shuck? Layered Hash Shucking.* DEFCON Safe Mode, August 2020. – https://www.youtube.com/watch?v=OQD3qDYMyYQ – Zugriff am: 03.09.2021

[DJPJ05] Desmet, Lieven ; Jacobs, Bart ; Piessens, Frank ; Joosen, Wouter: Threat Modelling for Web Services Based Web Applications. In: Chadwick, David (Hrsg.) ; Preneel, Bart (Hrsg.): *Communications and Multimedia Security.* Boston, MA : Springer US, 2005. – ISBN 978–0–387–24486–0, S. 131–144

[EPS15] Evans, Chris ; Palmer, Chris ; Sleevi, Ryan: Public Key Pinning Extension for HTTP / RFC Editor. http://dx.doi.org/10.17487/RFC7469. Internet Requests for Comments, April 2015. – RFC 7469. – ISSN 2070–1721

[FHBS+99] Franks, John ; Hallam-Baker, Phillip ; Stewart, Lawrence C. ; Hostetler, Jeffery L. ; Lawrence, Scott ; Leach, Paul J. ; Luotonen, Ari: HTTP Authentication: Basic and Digest Access Authentication / RFC Editor. http://dx.doi.org/10.17487/RFC2617. Internet Requests for Comments, Juni 1999. – RFC 2617. – ISSN 2070–1721

[FHT15] Farrell, Stephen ; Hoffman, Paul E. ; Thomas, Michael: HTTP Origin-Bound Authentication (HOBA) / RFC Editor. http://dx.doi.org/10.17487/RFC7486. Internet Requests for Comments, März 2015. – RFC 7486. – ISSN 2070–1721

[FIR19] FIRST.Org, Inc: *Common Vulnerability Scoring System version 3.1.* Specification Document, Juni 2019. – https://www.first.org/cvss/v3-1/cvss-v31-specification_r1.pdf – Zugriff am: 20.08.2021

[FLW13] Forler, Christian ; Lucks, Stefan ; Wenzel, Jakob: *Catena: A Memory-Consuming Password-Scrambling Framework.* Cryptology ePrint Archive, Report 2013/525, 2013. – https://ia.cr/2013/525

[Goo21] Google Cloud Architecture Center: *Strategien und Techniken zur Ratenbegrenzung.* https://cloud.google.com/architecture/rate-limiting-strategies-techniques, August 2021. – Zugriff am: 12.08.2021

[Got21] Goto, Sam: *WebOTP API.* W3C Draft Community Group Report, April 2021. – https://wicg.github.io/web-otp/ – Zugriff am: 27.07.2021

[GPTY18] Genkin, Daniel ; Pachmanov, Lev ; Tromer, Eran ; Yarom, Yuval: *Drive-by Key-Extraction Cache Attacks from Portable Code.* Cryptology ePrint Archive, Report 2018/119, 2018. – https://ia.cr/2018/119

[Han15] Hansen, Tony: SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms / RFC Editor. http://dx.doi.org/10.17487/RFC7677. Internet Requests for Comments, November 2015. – RFC 7677. – ISSN 2070–1721

[HB18] Harsha, Benjamin ; Blocki, Jeremiah: *Just in Time Hashing*. Cryptology ePrint Archive, Report 2018/112, 2018. – https://ia.cr/2018/112

[HJL+21] Hodges, Jeff ; Jones, J.C. ; Lundberg, Emil ; Kumar, Akshay ; Jones, Michael: Web Authentication: An API for accessing Public Key Credentials - Level 2 / W3C. 2021. – W3C Recommendation. – https://www.w3.org/TR/2021/REC-webauthn-2-20210408/ – Zugriff am: 28.08.2021

[Hod21] Hodges, Jeff: Credential Management Level 1 / W3C. 2021. – W3C Working Draft. – https://w3c.github.io/webappsec-credential-management/ – Zugriff am: 31.08.2021

[Hor19] Hornby, Taylor: *Salted Password Hashing – Doing it Right*. https://crackstation.net/hashing-security.htm, Juni 2019. – Zugriff am: 19.06.2021

[Hun18] Hunt, Troy: *I've Just Launched "Pwned Passwords"V2 With Half a Billion Passwords for Download*. https://www.troyhunt.com/ive-just-launched-pwned-passwords-version-2/, 2018. – Zugriff am: 15.07.2021

[ith15] ithare.com: *Client-Plus-Server Password Hashing as a Potential Way to Improve Security Against Brute Force Attacks without Overloading the Server*. http://ithare.com/client-plus-server-password-hashing-as-a-potential-way-to-improve-security-against-brute-force-attacks-without-overloading-server/, August 2015. – Zugriff am: 08.04.2021

[JBS15] Jones, Michael ; Bradley, John ; Sakimura, Nat: JSON Web Token (JWT) / RFC Editor. http://dx.doi.org/10.17487/RFC7519. Internet Requests for Comments, Mai 2015. – RFC 7519. – ISSN 2070–1721

[JH12] Jones, Michael ; Hardt, Dick: The OAuth 2.0 Authorization Framework: Bearer Token Usage / RFC Editor. http://dx.doi.org/10.17487/RFC6750. Internet Requests for Comments, Oktober 2012. – RFC 6750. – ISSN 2070–1721

[Lin21] Lindemann, Rolf: *FIDO Security Reference*. FIDO Alliance

Review Draft, Mai 2021. – https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-rd-20210525.pdf – Zugriff am: 30.08.2021

[LJR17] Llewellyn-Jones, David ; Rymer, G: Cracking PwdHash: A Bruteforce Attack on Client-side Password Hashing. (2017), April. http://dx.doi.org/10.17863/CAM.9058. – DOI 10.17863/CAM.9058

[LLK13] Laurie, Ben ; Langley, Adam ; Kasper, Emilia: Certificate Transparency / RFC Editor. http://dx.doi.org/10.17487/RFC6962. Internet Requests for Comments, Juni 2013. – RFC 6962. – ISSN 2070–1721

[McG13] McGrath, Michael: *Threat Modelling for Legacy Enterprise Applications*. Letterkenny Institute of Technology, August 2013. – https://research.thea.ie/handle/20.500.12065/1187 – Zugriff am: 20.08.2021

[Mel16] Melnikov, Alexey: Salted Challenge Response HTTP Authentication Mechanism / RFC Editor. http://dx.doi.org/10.17487/RFC7804. Internet Requests for Comments, März 2016. – RFC 7804. – ISSN 2070–1721

[MHN+05] M'Raihi, David ; Hoornaert, Frank ; Naccache, David ; Bellare, Mihir ; Ranen, Ohad: HOTP: An HMAC-Based One-Time Password Algorithm / RFC Editor. http://dx.doi.org/10.17487/RFC4226. Internet Requests for Comments, Dezember 2005. – RFC 4226. – ISSN 2070–1721

[MM18] Maqbali, Fatma A. ; Mitchell, Chris J.: Web Password Recovery: A Necessary Evil? http://dx.doi.org/10.1007/978-3-030-02683-7_23. In: *Proceedings of the Future Technologies Conference (FTC) 2018*. Springer International Publishing, Oktober 2018. – DOI 10.1007/978–3–030–02683–7_23, S. S. 324–341

[MRPM11] M'Raihi, David ; Rydell, Johan ; Pei, Mingliang ; Machani, Salah: TOTP: Time-Based One-Time Password Algorithm / RFC Editor. http://dx.doi.org/10.17487/RFC6238. Internet Requests for Comments, Mai 2011. – RFC 6238. – ISSN 2070–1721

[MSMWN10] Menon-Sen, Abhijit ; Melnikov, Alexey ; Williams, Nicolás ;

Newman, Chris: Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms / RFC Editor. http://dx.doi.org/10.17487/RFC5802. Internet Requests for Comments, Juli 2010. – RFC 5802. – ISSN 2070–1721

[Ngi19] Nginx Contributors: *Nginx Digest Authentication module.* https://github.com/atomx/nginx-http-auth-digest, 2019. – Zugriff am: 29.07.2021

[OWA21] OWASP CheatSheets Series Team: *Authentication Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html, Juni 2021. – Zugriff am: 13.07.2021

[PNB+18a] Popov, Andrei ; Nystrom, Magnus ; Balfanz, Dirk ; Langley, Adam ; Harper, Nick ; Hodges, Jeff: Token Binding over HTTP / RFC Editor. http://dx.doi.org/10.17487/RFC8473. Internet Requests for Comments, Oktober 2018. – RFC 8473. – ISSN 2070–1721

[PNB+18b] Popov, Andrei ; Nystrom, Magnus ; Balfanz, Dirk ; Langley, Adam ; Hodges, Jeff: The Token Binding Protocol Version 1.0 / RFC Editor. http://dx.doi.org/10.17487/RFC8471. Internet Requests for Comments, Oktober 2018. – RFC 8471. – ISSN 2070–1721

[Poh19] Pohlmann, Norbert: *Cyber-Sicherheit - Das Lehrbuch für Konzepte, Prinzipien, Mechanismen, Architekturen und Eigenschaften von Cyber-Sicherheitssystemen in der Digitalisierung.* 1. Wiesbaden : Springer Fachmedien Wiesbaden, 2019. – ISBN 978–3–658–25397–4

[Pow18] Powers, Adam: *FIDO TechNotes: The Truth about Attestation.* Juli 2018. – https://fidoalliance.org/fido-technotes-the-truth-about-attestation/ – Zugriff am: 28.08.2021

[Rab08] Rabkin, Ariel: Personal Knowledge Questions for Fallback Authentication: Security Questions in the Era of Facebook. In: *In SOUPS 2008: Proceedings of the 4th Symposium on Usable Privacy and Security, vol 23* (2008), Januar, S. 13–23. http://dx.doi.org/10.1145/1408664.1408667. – DOI 10.1145/1408664.1408667

[Rao21] Rao, Raja: *JSON Web Tokens (JWT) are Dangerous for User Sessions – Here's a Solution.* Redis Blog – https://redis.com/blog/json-web-tokens-jwt-are-dangerous-for-user-sessions/, Juni 2021. – Zugriff am: 04.09.2021

[RD08] Rescorla, Eric ; Dierks, Tim: The Transport Layer Security (TLS) Protocol Version 1.2 / RFC Editor. http://dx.doi.org/10.17487/RFC5246. Internet Requests for Comments, August 2008. – RFC 5246. – ISSN 2070–1721

[Ris16] Ristic, Ivan: *Is HTTP Public Key Pinning Dead?* https://blog.qualys.com/product-tech/2016/09/06/is-http-public-key-pinning-dead, September 2016. – Zugriff am: 11.07.2021

[RJM+05] Ross, Blake ; Jackson, Collin ; Miyake, Nicholas ; Boneh, Dan ; Mitchell, John: *Stronger Password Authentication Using Browser Extensions.* Proceedings of the 14th Usenix Security Symposium, 2005. – https://crypto.stanford.edu/PwdHash/pwdhash.pdf

[Sim96] Simpson, William A.: PPP Challenge Handshake Authentication Protocol (CHAP) / RFC Editor. http://dx.doi.org/10.17487/RFC1994. Internet Requests for Comments, August 1996. – RFC 1994. – ISSN 2070–1721

[SMA+13] Santesson, Stefan ; Myers, Michael ; Ankney, Rich ; Malpani, Ambarish ; Galperin, Slava ; Adams, Dr. C.: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP / RFC Editor. http://dx.doi.org/10.17487/RFC6960. Internet Requests for Comments, Juni 2013. – RFC 6960. – ISSN 2070–1721

[SPS11] *Kapitel* Authentifikations-Protokolle. In: Spitz, Stephan ; Pramateftakis, Michael ; Swoboda, Joachim: *Kryptographie und IT-Sicherheit - Grundlagen und Anwendungen.* Berlin Heidelberg New York : Springer-Verlag, 2011. – ISBN 978–3–834–88120–5

[SYAB15] Shekh-Yusef, Rifaat ; Ahrens, David ; Bremer, Sophie: HTTP Digest Access Authentication / RFC Editor. http://dx.doi.org/10.17487/RFC7616. Internet Requests for Comments, September 2015. – RFC 7616. – ISSN 2070–1721

[UY14] Uymatiao, Mariano L. ; Yu, William E.: Time-based OTP authentication via secure tunnel (TOAST): A mobile TOTP scheme using TLS seed exchange and encrypted offline keystore. (2014), April, S. 225–229. http://dx.doi.org/10.1109/ICIST.2014.6920371. – DOI 10.1109/ICIST.2014.6920371

[Wal20] Waller, Daniel: *Slow Down Online Guessing Attacks with Device Cookies*. OWASP Community Pages, Juli 2020. – https://owasp.org/www-community/Slow_Down_Online_Guessing_Attacks_with_Device_Cookies – Zugriff am: 28.07.2021

[Wat17] Watson, Mark: Web Cryptography API / W3C. 2017. – W3C Recommendation. – https://www.w3.org/TR/WebCryptoAPI/ – Zugriff am: 03.09.2021

[Wes16] West, Mike: Secure Contexts / W3C. 2016. – Candidate Recommendation. – https://www.w3.org/TR/secure-contexts/ – Zugriff am: 03.09.2021

[Whi21] Whited, Sam: Best practices for password hashing and storage / Internet Engineering Task Force. Work in Progress, April 2021. – Internet-Draft. – https://datatracker.ietf.org/doc/html/draft-ietf-kitten-password-storage-06 – Zugriff am: 16.08.2021

[Wil07] Williams, Nicolás: On the Use of Channel Bindings to Secure Channels / RFC Editor. http://dx.doi.org/10.17487/RFC5056. Internet Requests for Comments, November 2007. – RFC 5056. – ISSN 2070–1721

[Zor00] Zorn, Glen: Microsoft PPP CHAP Extensions, Version 2 / RFC Editor. http://dx.doi.org/10.17487/RFC2759. Internet Requests for Comments, Januar 2000. – RFC 2759. – ISSN 2070–1721

Abbildungsverzeichnis

2.1 Grundlegender Aufbau und Kommunikationspfade der Applikation 3
2.2 Zustandsübergänge der Protokollfunktionen 5
2.3 Modell zu Inhalt und Zusammensetzung eines X.509-Zertifikats . . 12
2.4 Beispiel zu Aufbau und Funktionsweise einer PKI-Architektur . . 13
2.5 Exemplarischer Ablauf eines *SSO*-Logins inklusive Anmeldung am *IDP* 16
2.6 Schematisches Vorgehen zum Aufbau einer Rainbow-Table 20
2.7 Referenz-Protokoll zur Registrierung ohne Sonderfallbehandlung . 29
2.8 Abläufe der Sitzungsverwaltung im Referenz-Protokoll 34

3.1 Vorgehen zur Wahl starker Passwörter ohne Passwortkenntnis durch den Server anhand des Prozesses zur Änderung von Anmeldedaten 42
3.2 Ablauf des Protokolls CHAP 65
3.3 Ablauf des Protokolls NTLMv1 66
3.4 Ablauf des Protokolls MS-CHAPv2 67
3.5 Ablauf des Protokolls zur Digest Access Authentication 67
3.6 Zusammensetzung der Entitäten des SCRAM-Protokolls 69
3.7 Ablauf einer Anmeldung mittels SCRAM-Protokoll 70
3.8 Mögliche Einbindung von SCRAM mit TLS Channel Binding in die Backend-Architektur mit *TLS*-Terminierung am Reverse-Proxy 73
3.9 Zeitliche Verteilung von Vorbereitung und Nachbereitung eines Data-Breaches bezüglich Hashing durch einen Angreifer 76

4.1 Protokoll zur optionalen, iterativen *MFA* mit begrenzten Versuchen 80
4.2 Mögliche Ergänzung einer MFA zum Referenz-Protokoll 83
4.3 Ablauf zur Aktivierung einer MFA zu einem Account 84
4.4 Beispiel einer SMS mit OTP im WebOTP-Format 88
4.5 Einsatz von Device-Cookies als zusätzliche Authentifizierungsfaktoren zur kontrollierten Umgehung von Rate-Limiting-Beschränkungen 94
4.6 Einsatz von Device-Cookies zur Mail-Verifikation mit Cookie-Prüfung durch den Server am Beispiel des Registrierungsprozesses . . 96
4.7 Einsatz von Device-Cookies zur Absicherung der iterativen MFA am Beispiel eines 2FA-Login-Vorgangs mit SCRAM und OTP . . 97
4.8 Architektur und Protokolle des FIDO2-Standards 106
4.9 Protokoll zur Faktor-Registrierung via FIDO2 107

5.1 *ERM* der Datenhaltung zur Referenz-Implementierung 117
5.2 *ERM* der essenziellen Konfiguration zur Referenz-Implementierung 117
5.3 Architektur und Technologien der Referenz-Implementierung . . . 118
5.4 Ablauf eines signaturgestützten Prozesses zur Software-Aktualisierung 120
5.5 Vergleich von Rendering-Typen für browsergestützte Applikationen 121
5.6 Ablauf der Überprüfung zur Passwortstärke ohne Passwort-Exponierung an den Server mithilfe des Datensatzes von „Have I Been Pwned“ 122
5.7 Vergleich der Hashing-Leistung verschiedener Endgeräte 131
5.8 Benchmarking zum Request-Durchsatz des Servers 133

A.1 Übersicht zu vorgestellten Verfahren und betrachteten Problemen 166

B.1 Screenshots zur Registrierung 168
B.2 Screenshots zur Anmeldung 168
B.3 Screenshots zur Zurücksetzung der Anmeldedaten 169
B.4 Screenshots zur Hauptbereich für angemeldete Nutzer 169
B.5 Screenshots zur Profilseite 170
B.6 Screenshots zu MFA-Einstellungsoptionen 170
B.7 Screenshots zur Registrierung eines TOTP-Faktors 171
B.8 Screenshots zur Registrierung eines FIDO2-Faktors 171
B.9 Screenshots zu weiteren Einstellungsoptionen 172
B.10 Screenshots zur MFA-Anmeldung 172

Tabellenverzeichnis

2.1 Einteilung der Risikobewertung anhand ermittelter Score-Werte . 11

3.1 Risiko durch Angreifer mit DB-Einblick ohne Peppers 43
3.2 Risiko durch Angreifer mit Einblick in DB und Peppers 43
3.3 Risiko der Übertragung von Passwörtern und Hashwerten via TLS an den Server 43
3.4 Risiko einer Authentifizierung nach Challenge-Response via TLS . 43
3.5 Risiko von ausschließlichem Client-Hashing 44
3.6 Risiko von Client-Hashing mit Server-Hashing 44
3.7 Risiko der Überlastung für langsames Server-Hashing 44
3.8 Risiko der Überlastung für schnelles Server-Hashing 44
3.9 Risiko zur Umgehung der Passwort-Regeln 44
3.10 Risiko zur Übertragung von Passwort-Hashpräfixen 44
3.11 Protokollvarianten zur Risiko-Betrachtung von clientseitigem Hashing 58
3.12 Risiko für CP1 zum Angriff mit vorberechneten Hashwerten . . . 58
3.13 Risiko für CP2 zum Angriff mit vorberechneten Hashwerten . . . 59
3.14 Risiko für CP3 durch DDOS-Attacke gegen die Pepper-Ausgabe . 60
3.15 Risiko für CP3 zum Angriff mit vorberechneten Hashwerten . . . 60
3.16 Risiko für CP3 zum Account-Stalking 60
3.17 Risiko für CP3 durch Account-Schwemme 60
3.18 Risiko für CP3 zur Timing-Attacke gegen die Pepper-Ausgabe . . 61
3.19 Risiko für CP3 zum MITM gegen die Pepper-Übertragung 61
3.20 Risiko für CP4 zur Timing-Attacke gegen die Pepper-Ausgabe . . 61
3.21 Risiko für CP4 durch Angreifer mit Einblick in die DB 61
3.22 Risiko der Server-Überlastung mittels DDOS-Attacken durch TLS-Einsatz 75
3.23 Risiko für CRAM zur Server-Überlastung mittels DDOS-Attacken 75
3.24 Risiko der Kompromittierung des Server-Zertifikats 75
3.25 Risiko der Kompromittierung eines Accounts 75
3.26 Risiko für SCRAM durch Angreifer mit DB-Einblick ohne Peppers 76
3.27 Risiko für SCRAM durch Angreifer mit DB-Einblick und Peppers 76

4.1 Risiko zum Phishing für Passwörter in Webapplikationen 98
4.2 Risiko zum Phishing für OTPs in Webapplikationen 98
4.3 Risiko zum Belauschen von OTP-Eingaben 99
4.4 Risiko für TOTP zum Replay alter OTP-Werte 99

4.5 Risiko zu OTPs für DB-Leak ohne Pepper 100
4.6 Risiko zu OTPs für DB-Leak mit Pepper 100
4.7 Risiko einer Replay-Attacke via Faktor-Downgrade 111
4.8 Risiko eines DB-Leaks mit Public-Keys 111
4.9 Risiko zur Schlüsselextraktion aus dem RAM 112
4.10 Risiko zur Schlüsselextraktion von der Festplatte 112
4.11 Risiko von FIDO2 bei kompromittiertem Client ohne Token Binding 112
4.12 Risiko von FIDO2 bei kompromittiertem Client mit Token-Binding 112
4.13 Risiko zum Origin-Spoofing mit FIDO2 via Desktop-Anwendung . 113
4.14 Risiko zum Brute-Force-Angriff auf Authenticators ohne UP . . . 113
4.15 Risiko von durch den Hersteller kompromittierten Security-Tokens 114
4.16 Risiko zum Nutzer-Tracking via präparierter Security-Tokens . . . 114

5.1 Übersicht zu eingesetzten Tokens und deren Eigenschaften 118
5.2 Möglichkeiten zur Wahl der Präfixlänge zur k-Anonymität 123
5.3 Benchmarking zur Hashing-Leistung von Webbrowsern gegenüber nativer Ausführung 129
5.4 Benchmarking zur Hashing-Leistung von Webbrowsern in Abhängigkeit vom Speicherbedarf für Argon2 130
5.5 Ausgewählte Endgeräte zum Testen der Hashing-Leistung 130
5.6 Zum Test der Einsatzfähigkeit evaluierte Security-Tokens 134
5.7 Einsatzfähigkeit von Security-Tokens via FIDO2 für ausgewählte Webbrowser auf unterschiedlichen Betriebssystemen 135

C.1 Dokumentation der REST-API 173

Quelltextverzeichnis

2.1 Bearbeitung einer Registrierungsanfrage durch den Server 30
2.2 Anweisungssequenz zur Anmeldung mit Fehlerbehandlung 31

3.1 Serverseitige Anweisungssequenz zur Anmeldung mit Salt-Rotation 51
3.2 Serverseitige Anweisungssequenz zur zeitsicheren Konstruktion authentifizierter Client-Pepper 57

5.1 Beispieleinträge aus dem Datensatz von „Have I Been Pwned“ ergänzt um die Passwort-Klartexte 122

Symbolverzeichnis

$\|\|$	Konkatenationsoperator
$\lfloor\ \rfloor$	Abrundung auf die nächst kleinere Ganzzahl
$\oplus$	Der XOR-Operator (Exklusive Oder-Verknüpfung)
$cost$	Der Kostenparameter für eine langsame, kryptographische Hashfunktion
$csprng$	Eine Generator-Funktion für kryptografisch-sichere Pseudo-Zufallszahlen
enc_{sym}	Symmetrische Verschlüsselungsfunktion
$hash$	Eine kryptografische Hashfunktion
$\|hash\|$	Die Länge des durch die Hashfunktion generierten Ausgabewertes in Bits
$hash^{fast}$	Eine schnelle kryptografische Hashfunktion
$hash^{slow}$	Eine langsame kryptografische Hashfunktion
$hmac$	Eine Funktion zur symmetrischen Signatur
kdf	Eine Schlüsselableitungsfunktion
$mail$	Die zu einem Account hinterlegte E-Mail-Adresse
$pepper$	Ein hoch-entropischer, geheimer Zufallswert, der in das Hashing von Passwörtern einbezogen wird
pw	Das Klartext-Passwort zu einem Account
$\|pw\|$	Die Passwortlänge in Zeichen
$salt$	Ein hoch-entropischer Zufallswert, der in das Hashing von Passwörtern einbezogen wird
$serial$	Funktion zur Serialisierung der übergebenen Datenstruktur in eine Zeichenkette
$serial^{-1}$	Funktion zur Deserialisierung einer Datenstruktur aus einer Zeichenkette
uid	Die eindeutige, identifizierende Benutzerkennung

Hinweise

Wird einem Symbol das Subskript „cli“ angehangen, findet eine clientseitige Verarbeitung statt. Entsprechend beschreibt das Subskript „svr“ eine Verarbeitung durch den Server.
Für Größen-Angaben von Datenmengen in Bytes werden die Binärpräfixe (mit einem Umrechnungsfaktor von 1024) verwendet: Kibibyte (KiB), Mebibyte (MiB), Gibibyte (GiB), Tebibyte (TiB), Pebibyte (PiB) etc.

Abkürzungsverzeichnis

1FA	Einfaktor-Authentifizierung
2FA	Zweifaktor-Authentifizierung
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BLE	Bluetooth Low Energy
CA	Certificate Authority
CFFI	C Foreign Function Interface
CHAP	Challenge Handshake Authentication Protocol
CPP	Client Puzzle Protocol
CPU	Central Processing Unit
CRAM	Challenge Response Authentication Mechanism
CRL	Certificate Revocation List
CSPRNG	Cryptographically Secure Pseudo-Random Number Generator
CSR	Certificate Signing Request
CTAP	Client to Authenticator Protocol
CVSS	Common Vulnerability Scoring System
DB	Datenbank
DBMS	Datenbankmanagementsystem
DDOS	Distributed Denial Of Service
DNS	Domain Name System
DOS	Denial Of Service
DREAD	Damage, Reproducibility, Exploitability, Affected-users, Discoverability
E2EE	End-To-End Encryption
ERM	Entity Relationship Model
FIDO	Fast Identity Online
GPG	GNU Privacy Guard

HKDF HMAC-based KDF
HMAC Keyed-Hash MAC
HOBA HTTP Origin-Bound Authentication
HOTP HMAC-based OTP
HPKP HTTP Public Key Pinning
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Secure HTTP

IDP Identity Provider
IoT Internet of Things
IP Internet Protocol
IPSec IP Security

JIT Just In Time
JSON JavaScript Object Notation
JWT JSON Web Token

KBKDF Key-Based KDF
KDF Key Derivation Function

LDAP Lightweight Directory Access Protocol
LRU Least Recently Used

M2M Machine-to-Machine
MAC Message Authentication Code
MD5 Message-Digest Algorithm 5
MFA Mehrfaktor-Authentifizierung
MITM Man In The Middle
mTLS Mutual TLS

NAT Network Address Translation
NFC Near Field Communication
NTLM NT LAN Manager

OCSP Online Certificate Status Protocol
OIDC OpenID Connect

OSINT Open Source Intelligence
OTP One Time Password
OWASP Open Web Application Security Project

PAKE Password-Authenticated Key Agreements
PAP Password Authentication Protocol
PASTA Process for Attack Simulation and Threat Analysis
PBKDF2 Password-Based Key Derivation Function 2
PHC Password Hashing Competition
PIN Personal Identification Number
PIV Personal Identity Verification
PKI Public Key Infrastructure
PoLP Principle of Least Privilege
PoW Proof of Work
PPP Point-to-Point Protocol
PWA Progressive Web App

QR Quick-Response

RAM Random-Access Memory
RBA Risikobasierte Authentifizierung
REST Representational State Transfer
RP Relying Party

SAML Security Assertion Markup Language
SCRAM Salted Challenge Response Authentication Mechanism
SHA Secure Hash Algorithm
SJCL Stanford Javascript Crypto Library
SMS Short Message Service
SMTP Simple Mail Transfer Protocol
SPA Single-Page Application
SQL Structured Query Language
SQLi SQL Injection
SSH Secure SHell
SSO Single Sign-On
STRIDE Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of Privilege

TLS Transport Layer Security
TMTO Time-Memory Tradeoff
TOAST Time-based OTP Authentication via Secure Tunnel
TOCTOU Time-Of-Check to Time-Of-Use
TOFU Trust On First Use
TOM Technische und Organisatorische Maßnahmen
TOTP Time-based OTP
TPM Trusted Platform Module
TRL Token Revocation List

U2F Universal 2nd Factor
UA User Agent
UI User Interface
UP User Presence
URI Uniform Resource Identifier
URL Uniform Resource Locator
USB Universal Serial Bus
UUID Universally Unique Identifier
UV User Verification
UX User Experience

VPN Virtual Private Network
VU Virtual User

WASM WebAssembly
WebAuthn Web Authentication API

Glossar

Account-Stalking Eine Form des Cyberstalkings, bei dem der Angreifer die öffentlich zu einem Account einsehbaren oder zugänglichen Informationen auswertet, um unerkannt Informationen zum Accountinhaber erlangen zu können.

Application Programming Interface (API) Eine definierte Schnittstelle, die eine Kommunikation nach festgelegtem Protokoll zwischen zwei entkoppelten Programmkomponenten ermöglicht.

Bearer-Token Eine Datenstruktur, die durch den Server für einen Client ausgestellt und vom Client zur Authentifizierung am Server vorgelegt wird.

Caching Das temporäre Vorhalten häufig abgerufener Inhalte aus Gründen der Performancesteigerung zu lasten des zusätzlichen Speicherbedarfs.

Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) Ein Generator, der pseudo-zufällig Zahlen erzeugt, deren Verteilung als hinreichend zufällig genug angesehen wird, um sie für kryptographische Zwecke zu nutzen.

Data Breach Eine versehentliche Veröffentlichung von Datenbeständen führt zur unerlaubten Einblicknahme durch Dritte.

Deployment Als Deployment wird ein laufendes Softwaresystem bezeichnet, das auf einer dazu vorgesehenen Infrastruktur oder Ausführungsumgebung betrieben wird.

Diffie-Hellman-Key Der Diffie-Hellman-Schlüsselaustausch erlaubt es zwei Kommunikationsteilnehmern über einen öffentlichen, abhörbaren Kanal einen geheimen Schlüssel in Form einer Zahl zu vereinbaren. Dieser Diffie-Hellman-Schlüssel ist für Beobachter des Kanals anhand der ausgetauschten Informationen nicht trivial rekonstruierbar. Das Diffie-Hellman-Verfahren kann über Galoisfeldern oder elliptischen Kurven konstruiert werden.

Docker Open-Source-Software zur Containerisierung von Anwendungen, um diese isoliert voneinander ausführen zu können. Die Applikationen werden hierbei in Container verpackt, die vom Hauptsystem abgeschottete Laufzeitumgebungen erzeugen. Die Ressourcen des Hauptsystems werden unter den Containern aufgeteilt. Die Bereitstellung von Software wird durch die Verwendung von Docker vereinfacht, da die Software in einem einheitlichen Format, unabhängig verwendeter Technologien, paketiert wird.

Downgrade-Attacke Das betroffene Protokoll erlaubt einen Wechsel von den standardmäßig eingesetzten, sicheren Verfahren auf unsichere Verfahren, z. B. aus Kompatibilitätsgründen. Ein Angreifer kann u. a. die Verschlüsselung aufhebeln, indem er einen Wechsel zu einem unsicheren Verschlüsselungsverfahren erzwingt, z. B. als *MITM*.

Eavesdropper Ein passiver Angreifer, der einen Informationsaustausch belauscht oder aufzeichnet.

Epoch Eine Referenzgröße für Zeitangaben. Sie beschreibt die Anzahl der seit dem 01.01.1970 in Weltzeit vergangenen Sekunden oder Millisekunden ohne Verzerrungen wie Schaltsekunden.

Garbage-Collection Automatische Speicherbereinigung, die zur Laufzeit eines Programms in Intervallen nicht mehr benötigte Speicherbereiche sucht und gefundene freigibt.

Hashing Verwendung einer kryptografischen Funktion zur surjektiven Abbildung einer Eingabedatensequenz beliebiger Länge auf eine Zielmenge fester Länge. Letztere wird Hashwert genannt.

JavaScript Programmiersprache, die zur Gruppe der Skriptsprachen gehört. Die Sprache wurde entwickelt, um dynamische Websites mit Benutzerinteraktionen zu ermöglichen. Mittels JavaScript können Inhalte von Websites verändert, nachgeladen oder generiert werden.

JSON Web Token (JWT) Ein Format für client-gestützte *Bearer-Tokens*.

Keylogger Programm oder Hardware, welches Nutzereingaben am Endgerät, z. B. von der Tastatur, unbemerkt aufzeichnet und speichern oder direkt an den Angreifer versenden kann.

Leetspeak Techniken zur Verfremdung von Textzeichen durch Ersetzung mit optisch ähnlichen Zeichen.

Lightweight Directory Access Protocol (LDAP) Protokoll und Dienst zur Verwaltung von Nutzern und Rollen. Es kann zur Authentifizierung und Autorisierung von Anwendern gegenüber angebundenen Applikationen genutzt werden.

Man In The Middle (MITM) Ein aktiver Angreifer, der sich in die Kommunikationsstrecke zwischen den Teilnehmern eingebracht hat, um den Informationsaustausch manipulieren zu können.

Micro-Service Eine funktional eigenständige Komponente, die eine gewisse fachliche Logik erfüllt. Die Kapselung fachlicher Logikgruppen zu separaten Architekturbestandteilen bietet gegenüber einer monolithischen Architektur die Möglichkeit, eine bessere Übersicht über Kommunikations- und Datenaustauschsbeziehungen der einzelnen Services zu erhalten.

Mutex Verfahren zur Etablierung eines wechselseitigen Ausschlusses bzw. Locks. Dadurch werden parallele Zugriffe auf eine Ressource unterbunden, sodass eine anfragende Partei die Ressource ausschließlich, ohne Seiteneffekte durch andere Parteien nutzen kann.

Nginx Ein performanter, quelloffener *Webserver*, der u. a. als Reverse-Proxy und zum Ausliefern statischer Dateien nutzbar ist.

Nonce Ein Wert, der nur einmalig im Verwendungskontext einzusetzen ist. Dies kann beispielsweise in Form einer Zahl, die nach jeder Nutzung inkrementiert wird, realisiert werden.

Opaque Eine mittels *CSPRNG* gewählte, zufällige Bitsequenz.

Origin-Spoofing Täuschungsversuch, bei dem der Angreifer dem Nutzer vorspielt mit einer legitimen Quelle zu interagieren, obwohl das Opfer mit dem System des Angreifers interagiert. Eine solche Quelle ist üblicherweise eine Domain, Webseite oder E-Mail.

Pass-the-Hash-Angriff Angriff bei dem der Angreifer sich mithilfe des Hashwertes eines Passwortes gegenüber dem System authentifizieren kann, ohne das eigentlich notwendige Passwort kennen zu müssen.

PostgreSQL Freies, relationales Datenbanksystem. Der Zugriff auf den Datenbankdienst erfolgt über eine Server-Schnittstelle und die Kommunikation mit dem *Datenbankmanagementsystem (DBMS)* findet mittels *Structured Query Language (SQL)* statt.

Privilege Escalation Ausnutzung einer Schwachstelle, um höhere Rechte erlangen zu können, als dem Angreifer eigentlich durch das System zugestanden werden.

Progressive Web App (PWA) Eine *Webanwendung*, die als native Applikation fungieren kann. Hierdurch wird eine bessere Integration in das ausführende System erzielt.

Python Eine dynamisch typisierte, speicher-sichere, asynchrone Programmiersprache, die zur Familie der Skriptsprachen gehört.

Rainbow-Table Eine Datenstruktur zur Speicherung vorberechneter Hashwerte inklusive Ausgangswerten. Rainbow-Tables werden zum Finden eines Passwortes zu einem gegebenen Hashwert genutzt. Der Speicherbedarf ist geringer als bei Speicherung sämtlicher enthaltener Hashwerte. Im Gegenzug ist der Zeitaufwand zum Finden des Ausgangswertes erhöht.

Redis Eine Datenbank, die die Daten zur Steigerung der Performance im Arbeitsspeicher hält.

Representational State Transfer (REST) Paradigma zum Design von Schnittstellen in verteilten Systemen. REST ist mittels *HTTP* realisierbar und findet u. a. bei Webservices Anwendung. Ziel ist die Schaffung einer einheitlichen, zustandslosen, lose gekoppelten Schnittstelle, die Daten auf Basis von Ressourcen untergliedert und ordnet.

Reverse-Proxy-Server Server-Anwendung, die Zugriffe auf ein Netzwerk mittels eines zentralen Servers bündelt. Dieser Server erhält alle Anfragen an das Netzwerk und leitet diese anhand festgelegter Routing-Regeln an weitere

Server innerhalb des Netzwerks weiter. Diese beantworten die Anfrage und senden ihre Antwort über den Reverse-Proxy-Server zurück an das Frontend. Eine derartige Backend-Struktur bietet den Vorteil, dass vom inneren Netzwerkaufbau abstrahiert wird.

Roundtrip Ein Informationsaustausch im Netzwerk vom Sender zum Empfänger und zurück.

Sandbox Eine isolierte Ausführungsumgebung, die nur eine Durchführung limitierter Aktionen innerhalb der Umgebung gestattet. Dadurch wird eine Abschottung der Sandbox von der äußeren Umgebung erzielt, sodass eine mögliche Handlung in der Sandbox keine unbeabsichtigten Wechselwirkungen mit der äußeren Umgebung verursachen können soll.

Script-Kiddies Technisch unbedarfte Angreifer, die durch den Einsatz von ihnen zugänglichen Hacking-Tools Schaden anrichten können, ohne die Funktionsweise dieser Toolkits verstanden zu haben.

Secure SHell (SSH) Programm zur Herstellung eines verschlüsselten Tunnels zu einem entfernten Computersystem. Das gleichnamige Protokoll erlaubt nach Aufbau des Tunnels u. a. eine Terminaleingabe auf dem entfernten System, Portweiterleitungen vom oder zum entfernen System und Dateitransfers zwischen lokalem und entferntem System.

Server-Relief Größtmögliche Verringerung der Last des Serversystems durch Verschiebung der durch eine Applikation im Betrieb erzeugten Last zu den Clientsystemen.

Single Sign-On (SSO) Der Login bei einem Dienst wird für andere Anwendungen übernommen, sodass an letzteren eine separate Anmeldung mit Nutzerinteraktion entfällt.

Single-Page Application (SPA) Eine spezielle Form einer *Webanwendung*, bei der sämtliche Seiteninhalte beim ersten Aufruf der Webseite geladen werden. Anschließende Navigation auf der Webseite findet nur innerhalb des geladenen *HTML*-Dokuments statt, wobei weitere Inhalte dynamisch nachgeladen werden können. Durch ein *Caching* der Seiteninhalte ist die Anwendung potenziell auch offline verwendbar.

Singleton Entwurfsmuster, bei dem von einer Entität nur eine Instanz zur Laufzeit existiert, welche an mehreren Stellen im Programmablauf verwendet werden kann.

SQL-Injection Angriff auf ein System durch Einschleusen von Anweisungen für die Datenbank-Kommunikation an Stellen, wo durch den Nutzer eigentlich nur Wertparameter anstatt Befehlen eingegeben werden dürften.

Svelte Ein Applikations-Framework zur Erstellung von Webseiten bzw. Webanwendungen als *SPAs* mithilfe der Programmiersprache *JavaScript*.

Traefik Ein quelloffener *Reverse-Proxy-Server* für *Webanwendungen*, der insbesondere für den Einsatz in *Docker*-Umgebungen ausgelegt ist.

Unicode Ein Standard zur Kodierung von Textzeichen, welcher einen hohen Zeichenumfang aufweist. Ziel ist die Schaffung eines Korpus, der weltweit verwendete Zeichensysteme umfasst. Dadurch müssen keine unterschiedlichen Kodierungen eingesetzt werden, die zudem inkompatibel zueinander sein können.

Uniform Resource Locator (URL) Ein eindeutiger Identifikator und gleichzeitig Adresse einer Ressource in einem Netzwerk. Sie setzt sich aus verwendetem Protokoll, Hostname, Pfadangaben sowie weiteren Parametern zusammen.

User-Agent Das Programm auf dem Client-System, welches der Anwender nutzt, um mit einem Server-System zu kommunizieren.

Virtual Private Network (VPN) Ein Kommunikationsnetz, das nicht öffentlich zugänglich ist. Es ermöglicht den Teilnehmern, die sich nicht im selben physischen Netzwerk befinden müssen, die direkte Kommunikation, indem das Netzwerk als Transportmedium zwischen verschiedenen Netzen agiert.

Webanwendung Applikationsform, die eine Client- und Server-Komponente umfasst. Der Client-Anwendungsteil wird im *Webbrowser* des Endanwenders ausgeführt. Für die zentrale Datenverarbeitung und Datenhaltung wird ein Serversystem genutzt. Da die Ausführung des Frontends im *Webbrowser* stattfindet, ist eine zusätzliche Software-Installation im Allgemeinen nicht notwendig.

Webbrowser Client zur Darstellung von Webinhalten.

Webclient Die Komponente einer *Webanwendung*, die im Browser des Endanwenders ausgeführt wird.

Webserver Serverprogramm, das Inhalte über das Protokoll *HTTP* bereitstellt.

Zeilenmetrik Ein Verfahren zur Messung von Softwarekomplexität, das die Anzahl an Quellcodezeilen eines Programms oder Moduls betrachtet, um eine Bewertung der Komplexität des Quellcodes zu ermöglichen. Als Verfeinerung wird mitunter zwischen Programmzeilen und Kommentarzeilen unterschieden.

Anlagen

A Übersicht zu vorgestellten Verfahren und betrachteten Problemen

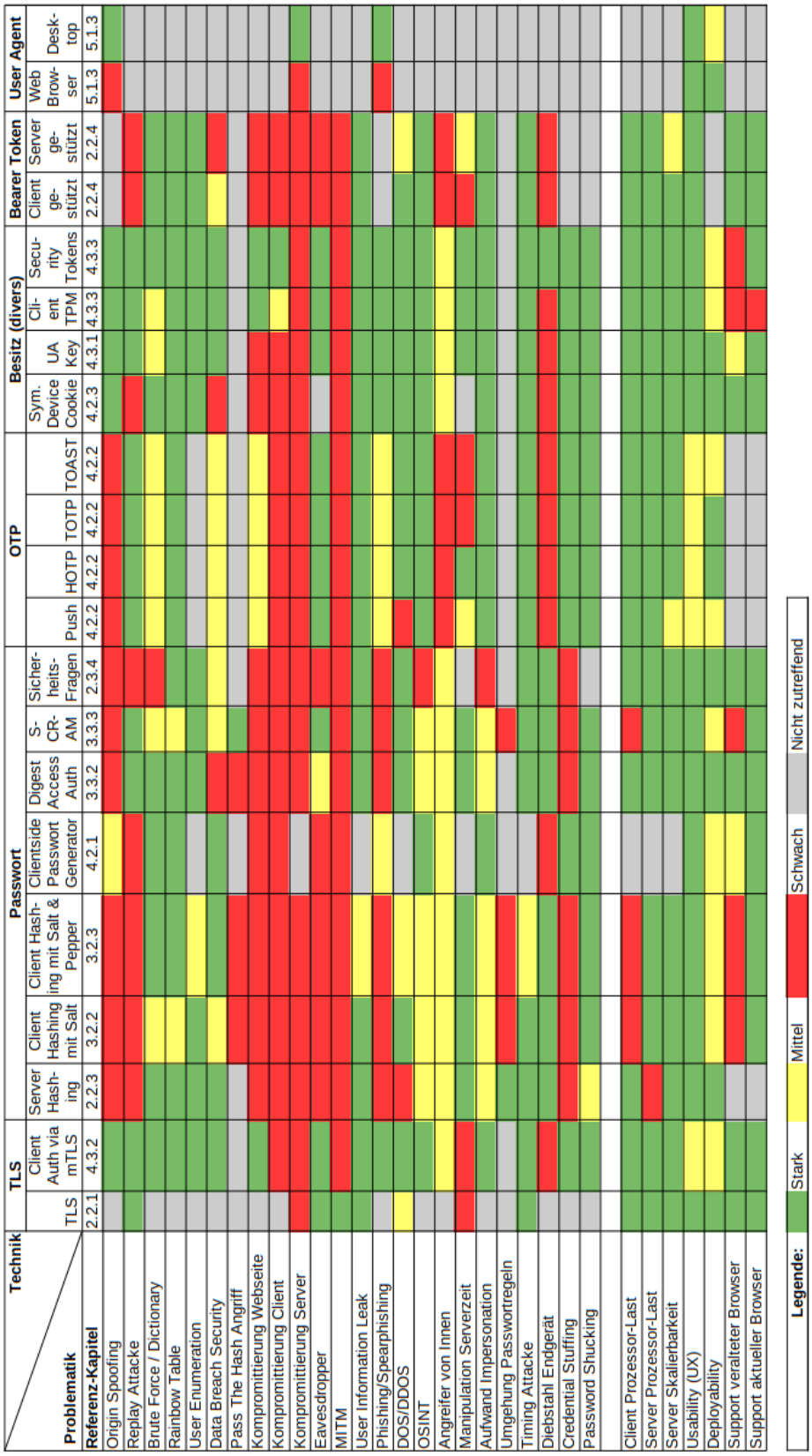


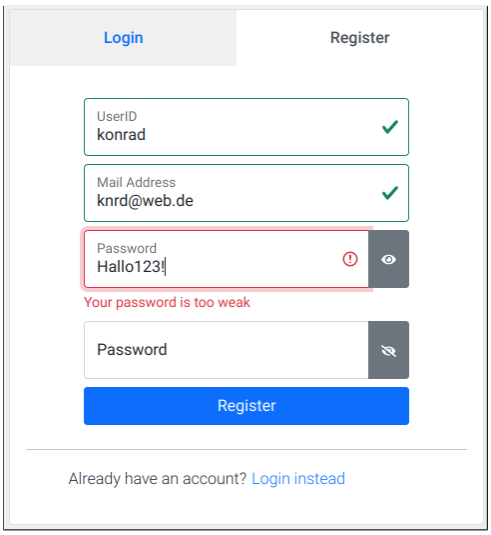
Abbildung A.1: Übersicht zu vorgestellten Verfahren und betrachteten Problemen
Quelle: Eigene Darstellung

Kurzbeschreibung der Angriffsszenarien

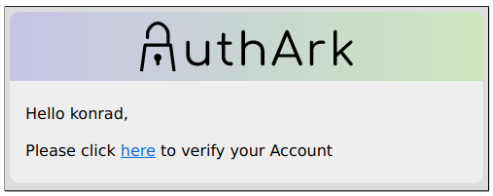
- **Origin Spoofing**: Vortäuschung einer anderen Webseite, um das Opfer zur Dateneingabe auf der falschen Webseite zu bewegen.
- **Replay Attacke**: Aufgezeichnete Kommunikation wird erneut eingespielt.
- **Brute Force / Dictionary**: Angriff durch Ausprobieren von möglichen, vermuteten Passwortwerten.
- **Rainbow Table**: Angriff mit vorberechneten Hashwerten.
- **User Enumeration**: Angreifer erfährt welche Accounts an einer Applikation registriert sind.
- **Data Breach Security**: Sicherheit im Falle eines Bekanntwerdens des Datenbestandes an unbefugte Dritte.
- **Pass The Hash Angriff**: Anmeldung mit dem Hashwert anstatt des originalen Passwortes möglich.
- **Kompromittierung Webseite**: Der Webclient der Applikation wird kompromittiert, wovon allerdings nicht der *UA* betroffen ist.
- **Kompromittierung Client**: Client-Software bzw. *UA* werden kompromittiert.
- **Kompromittierung Server**: Die Server-Software steht unter dem Einfluss Dritter, was Nutzer und ihre zur Applikation hinterlegten Daten bedroht.
- **Eavesdropper**: Passives Belauschen des Datenaustauschs zwischen Client und Server, siehe Kapitel 2.1.2.
- **MITM**: Aktives Belauschen und Manipulieren des Datenaustauschs zwischen Client und Server, siehe Kapitel 2.1.2.
- **User Information Leak**: Bekanntgabe privater Nutzerinformationen an unberechtigte Dritte.
- **Phishing / Spearphishing**: Siehe Kapitel 2.1.2.
- **DOS / DDOS**: Siehe Kapitel 2.1.2.
- **OSINT**: Angriff mit öffentlich einsehbaren Nutzerdaten, siehe Kapitel 2.1.2.
- **Angreifer von Innen**: Siehe Kapitel 2.1.2.
- **Manipulation Serverzeit**: Angreifer kann Einfluss auf die Systemzeit der Serversysteme nehmen.
- **Aufwand Impersonation**: Aufwand für einen Angreifer, um unter der Identität des Opfers an der Applikation agieren zu können.
- **Umgehung Passwortregeln**: Aufwand für einen Nutzer zur Missachtung der Passwortregeln.
- **Timing Attacke**: Angreifer zieht Rückschlüsse aus der Verarbeitungszeit des Servers für eine Anfrage.
- **Diebstahl Endgerät**: Möglichkeiten eines Angreifers bei voller Kontrolle für das Endgerät des Opfers.
- **Credential Stuffing**: Ausnutzung einer Wiederverwendung von Anmeldedaten bei verschiedenen Applikationen, siehe Kapitel 2.1.2.
- **Password Shucking**: Siehe Kapitel 2.2.3.

B Dokumentation des Webclients

Nachfolgend werden die Nutzungsabläufe der *IDP*-Applikation entsprechend Kapitel 5.1.1 mittels Screenshots visualisiert dargestellt.

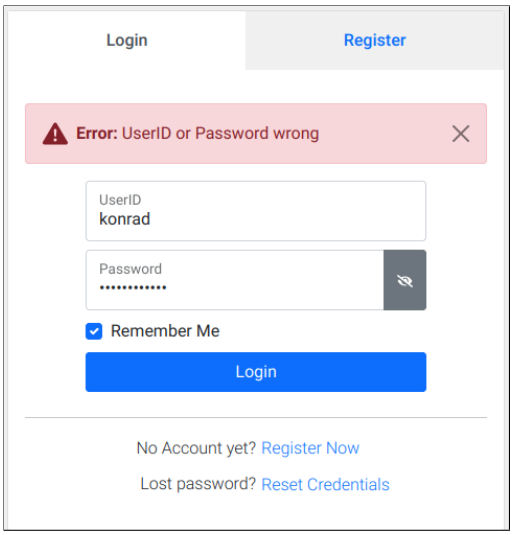


(a) Angabe Stammdaten



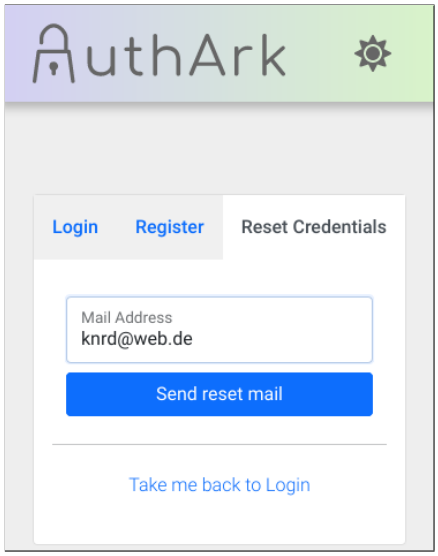
(b) Account-Verifikation via E-Mail

Abbildung B.1: Screenshots zur Registrierung
Quelle: Eigene Darstellung

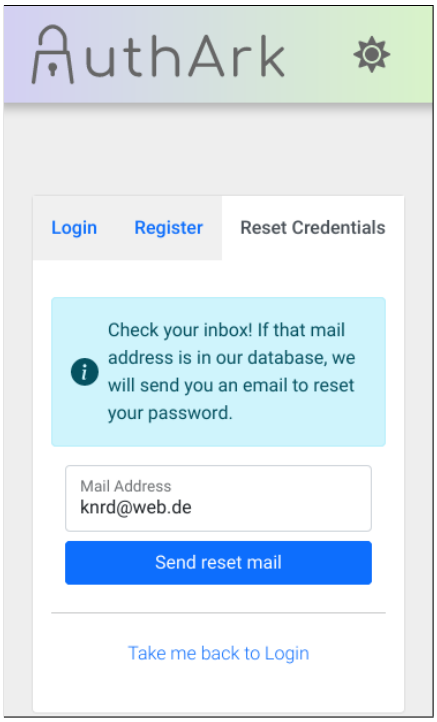


(a) Anmeldemaske mit Rückmeldung zu vorheriger Falscheingabe

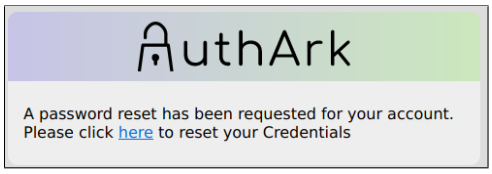
Abbildung B.2: Screenshots zur Anmeldung
Quelle: Eigene Darstellung



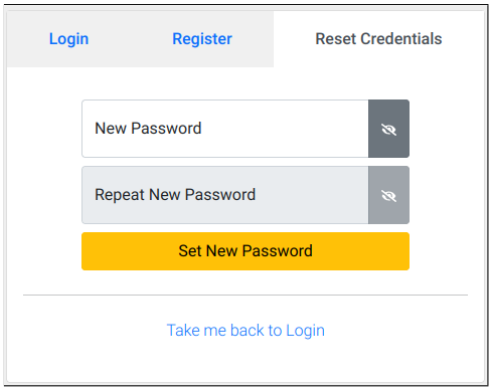
(a) Anforderung



(b) Antwort

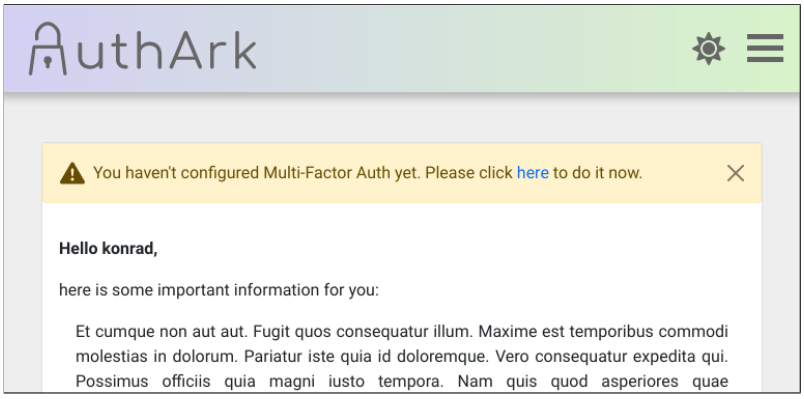


(c) Mail-Verifikation

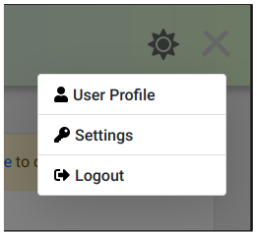


(d) Neuvergabe

Abbildung B.3: Screenshots zur Zurücksetzung der Anmeldedaten
Quelle: Eigene Darstellung

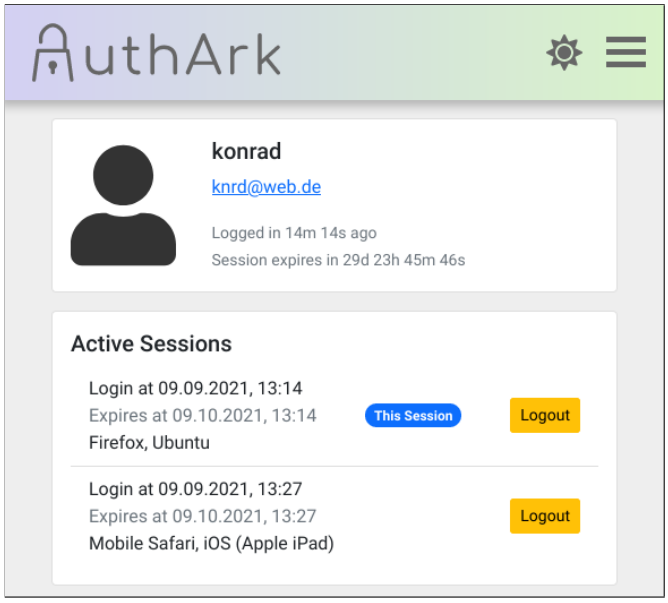


(a) Hauptseite



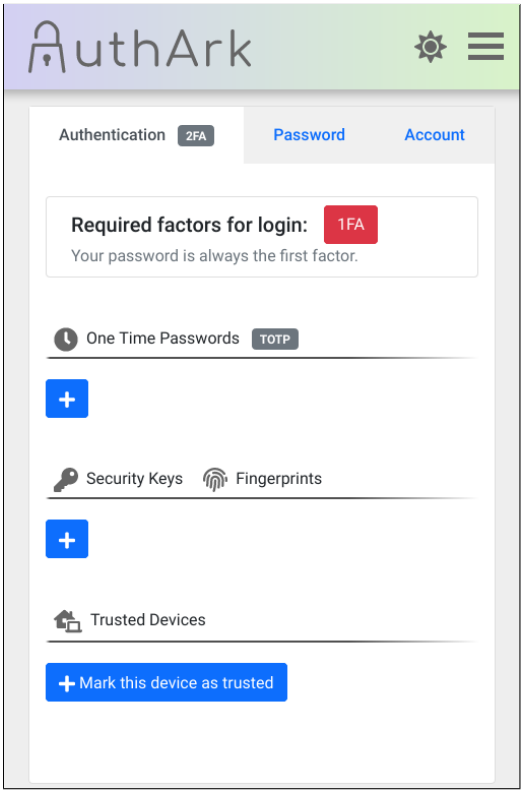
(b) Menüführung

Abbildung B.4: Screenshots zur Hauptbereich für angemeldete Nutzer
Quelle: Eigene Darstellung

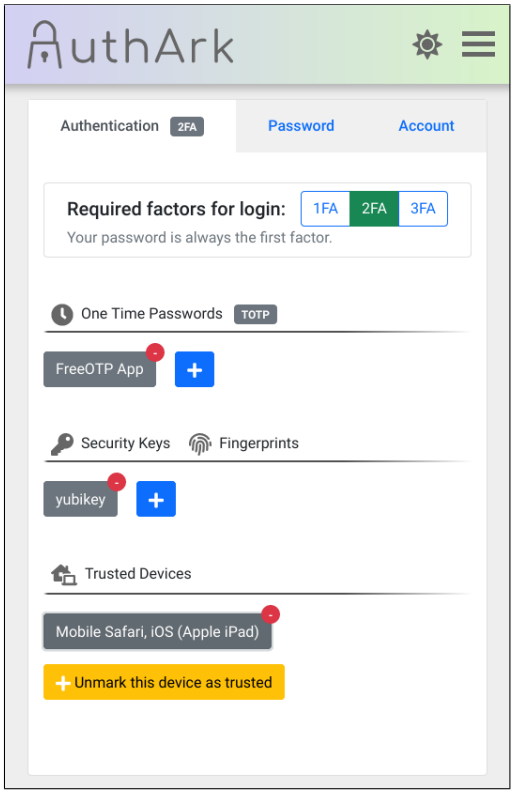


(a) Nutzerprofil und aktive Sitzungen

Abbildung B.5: Screenshots zur Profilseite
Quelle: Eigene Darstellung

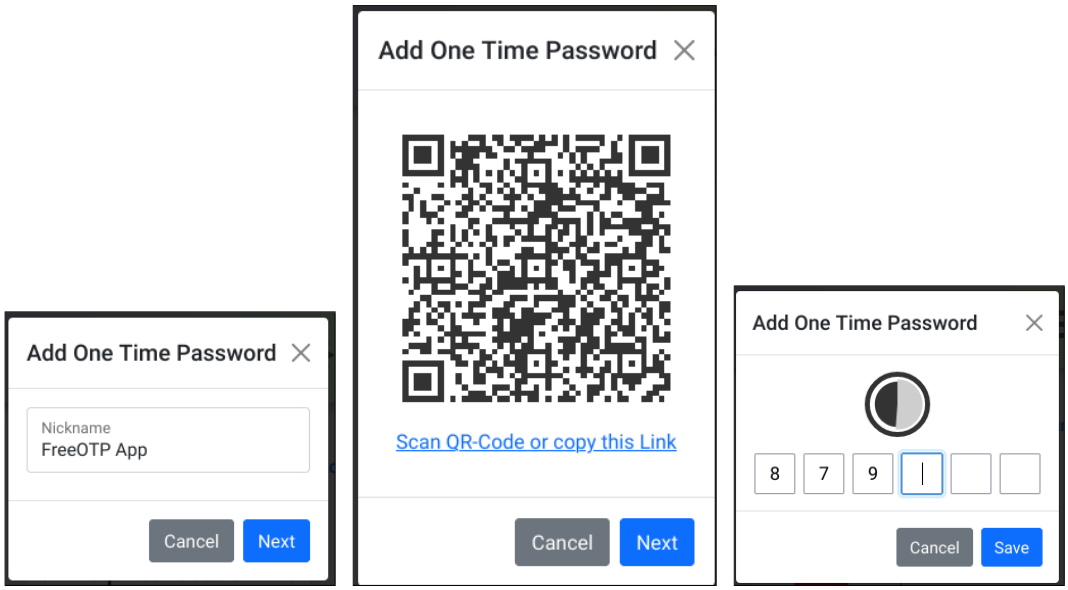


(a) 1FA ohne MFA-Konfiguration



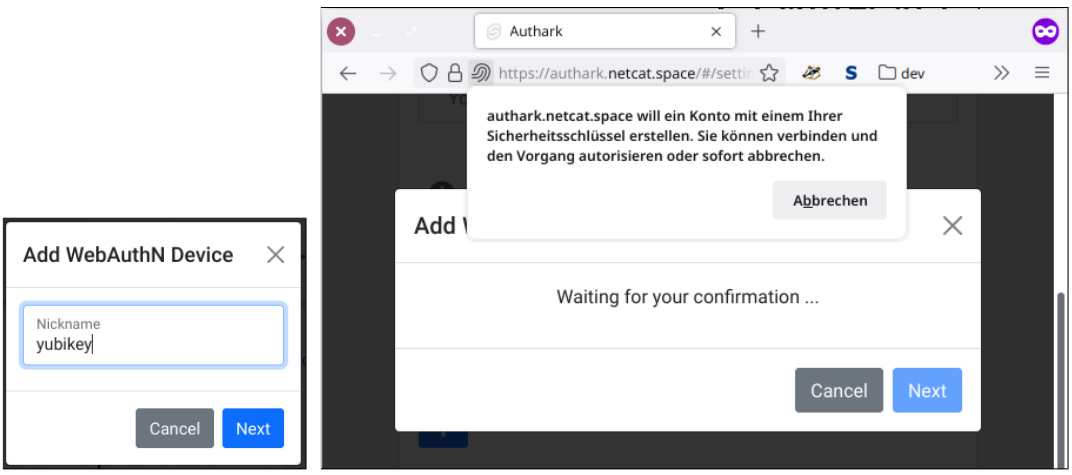
(b) 2FA mit hinterlegten MFA-Faktoren

Abbildung B.6: Screenshots zu MFA-Einstellungsoptionen
Quelle: Eigene Darstellung



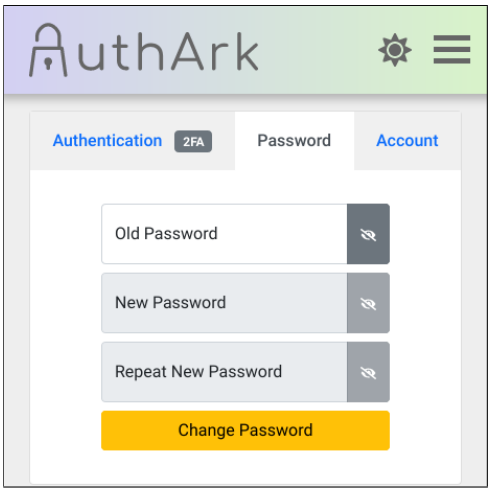
(a) Vergabe Nickname **(b)** Scan auf Zweitgerät **(c)** Verifikation

Abbildung B.7: Screenshots zur Registrierung eines TOTP-Faktors
Quelle: Eigene Darstellung

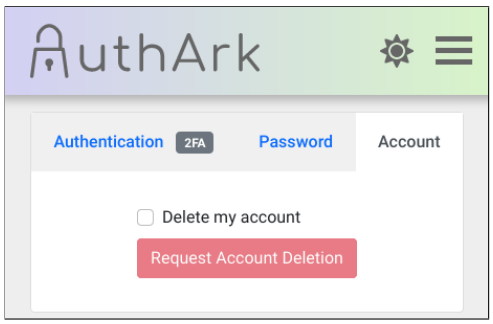


(a) Vergabe Nickname **(b)** Verifikation

Abbildung B.8: Screenshots zur Registrierung eines FIDO2-Faktors
Quelle: Eigene Darstellung

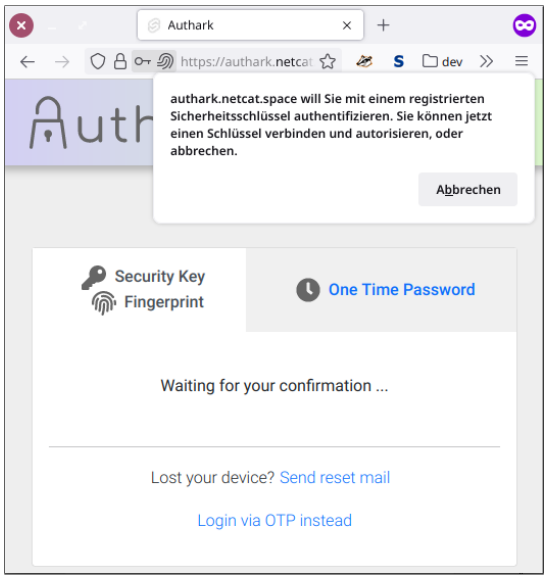


(a) Passwortänderung

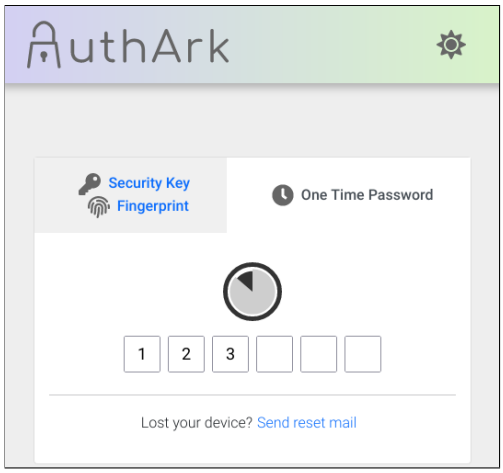


(b) Account-Löschung

Abbildung B.9: Screenshots zu weiteren Einstellungsoptionen
Quelle: Eigene Darstellung



(a) Via FIDO2



(b) Via TOTP

Abbildung B.10: Screenshots zur MFA-Anmeldung
Quelle: Eigene Darstellung

C Dokumentation der REST-API

Tabelle C.1 zeigt die Schnittstellendefinitionen, die eine Applikation nach *REST*-Paradigma mindestens implementieren muss, um den Funktionsumfang aus Kapitel 5.1.1 abbilden zu können.

Tabelle C.1: Dokumentation der REST-API

Method	Resource	Description	Session
Namespace: `/api/accounts`			
GET	`/<uid>/registered` Request Path: uid*(str)* ⟨200⟩ Response Body: registered*(bool)*	Check User-ID is registered	
GET	`/registration/scram` ⟨200⟩ Response Body: cost*(dict)*, public_salt*(bytes)*	Get SCRAM Registration Parameter	
POST	`/registration` Request Body: key_db*(bytes)*, key_svr*(bytes)*, mail*(str)*, uid*(str)*, verification_challenge*(bytes)* ⟨204⟩ Empty Response	Create Account	
PATCH	`/registration` Request Body: challenge_verifier*(bytes)*, mail_token*(str)* ⟨204⟩ Empty Response	Verify Account	
POST	`/credentials/reset` Request Body: mail*(str)*, verification_challenge*(bytes)* ⟨204⟩ Empty Response	Request Credentials Reset	
PUT	`/credentials/reset` Request Body: challenge_verifier*(bytes)*, key_db*(bytes)*, key_svr*(bytes)*, reset_token*(str)* ⟨204⟩ Empty Response	Finish Credentials Reset	
PUT	`/credentials` Request Body: key_db*(bytes)*, key_svr*(bytes)*, proof_cli*(bytes)*, proof_cli_old*(bytes)*, scram_token*(str)* ⟨204⟩ Empty Response	Change Credentials	×
POST	`/deletion` Request Body: verification_challenge*(bytes)* ⟨204⟩ Empty Response	Request Account Deletion	×
PUT	`/deletion` Request Body: challenge_verifier*(bytes)*, mail_token*(str)*	Authenticate Account Deletion	

Fortführung auf nächster Seite

Tabelle C.1 – *Fortführung*

Method	Resource	Description	Session
	⟨200⟩ Response Body (Enter MFA): state*(enum)*, mfa_token*(str)*, list mfa_kinds*(enum)*, mfa_webauthn_options*(dict)*, last_attempt_successful*(bool)* ⟨200⟩ Response Body (Account Deletion): state*(enum)*		
Namespace: /api/auth			
POST	`/1fa/scram`	Init Scram Login	
	Request Body: nonce_client*(bytes)*, uid*(str)* ⟨200⟩ Response Body: challenge*(bytes)*, challenge_old*(bytes)*, cost*(dict)*, cost_old*(dict)*, in_rotation_transition*(bool)*, salt*(bytes)*, salt_old*(bytes)*, scram_token*(str)*		
PUT	`/1fa/scram`	Finish Scram Login	
	Request Body: proof_cli*(bytes)*, proof_cli_old*(bytes)*, scram_token*(str)* ⟨200⟩ Response Body (Enter MFA): state*(enum)*, need_scram_update*(bool)*, server_sign*(bytes)*, mfa_token*(str)*, list mfa_kinds*(enum)*, mfa_webauthn_options*(dict)*, last_attempt_successful*(bool)* ⟨200⟩ Response Body (Session Creation): state*(enum)*, access_token*(str)*, refresh_token*(str)* ⟨200⟩ Response Body (Account Deletion): state*(enum)*		
POST	`/mfa/totp`	Login via Totp	
	Request Body: mfa_token*(str)*, totp_code*(str)* ⟨200⟩ Response Body (Continue MFA): state*(enum)*, mfa_token*(str)*, list mfa_kinds*(enum)*, mfa_webauthn_options*(dict)*, last_attempt_successful*(bool)* ⟨200⟩ Response Body (Session Creation): state*(enum)*, access_token*(str)*, refresh_token*(str)* ⟨200⟩ Response Body (Account Deletion): state*(enum)*		
POST	`/mfa/webauthn`	Login via Webauthn	
	Request Body: authenticator_data*(bytes)*, client_data_json*(bytes)*, credential_id*(bytes)*, mfa_token*(str)*, signature*(bytes)* ⟨200⟩ Response Body (Continue MFA): state*(enum)*, mfa_token*(str)*, list mfa_kinds*(enum)*, mfa_webauthn_options*(dict)*, last_attempt_successful*(bool)* ⟨200⟩ Response Body (Session Creation): state*(enum)*, access_token*(str)*, refresh_token*(str)* ⟨200⟩ Response Body (Account Deletion): state*(enum)*		
POST	`/mfa/ua-key`	Login via Browser Key	
	Request Body: factor_id*(uuid)*, mfa_token*(str)*, signature*(bytes)*		

Fortführung auf nächster Seite

Tabelle C.1 – *Fortführung*

Method	Resource	Description	Session
	⟨200⟩ Response Body (Continue MFA): state*(enum)*, mfa_token*(str)*, list mfa_kinds*(enum)*, mfa_webauthn_options*(dict)*, last_attempt_successful*(bool)* ⟨200⟩ Response Body (Session Creation): state*(enum)*, access_token*(str)*, refresh_token*(str)* ⟨200⟩ Response Body (Account Deletion): state*(enum)*		
Namespace: `/api/authfactors`			
GET	`/`	Get Factors and Settings	×
	⟨200⟩ Response Body: max_mfa_level*(int)*, current_mfa_level*(int)*, list of factors: kind*(enum)*, factor_id*(enum)*, nickname*(str)*		
PUT	`/settings`	Set Factors Settings	×
	Request Body: mfa_level*(int)* ⟨204⟩ Empty Response		
DELETE	`/<factor_id>`	Remove Factor	×
	Request Path: factor_id*(uuid)* ⟨204⟩ Empty Response		
POST	`/totp`	Begin Totp Registration	×
	⟨200⟩ Response Body: provisioning_uri*(str)*, totp_registration_token*(str)*		
PUT	`/totp`	Complete Totp Registration	×
	Request Body: current_otp*(str)*, nickname*(str)*, totp_registration_token*(str)* ⟨204⟩ Empty Response		
POST	`/webauthn`	Begin Webauthn Device Registration	×
	⟨200⟩ Response Body: registration_data*(dict)*, webauthn_registration_token*(str)*		
PUT	`/webauthn`	Complete Webauthn Device Registration	×
	Request Body: attestation_object*(bytes)*, client_data_json*(dict)*, nickname*(str)*, webauthn_registration_token*(str)* ⟨204⟩ Empty Response		
GET	`/ua-key`	Get Browser Key Registration Parameter	×
	⟨200⟩ Response Body: name*(enum)*, modulus*(int)*, exponent*(int)*, hash*(enum)*		
POST	`/ua-key`	Register Browser Key	×
	Request Body: public_key*(dict)* Request Header: `User-Agent`		

Fortführung auf nächster Seite

Tabelle C.1 – *Fortführung*

Method	Resource	Description	Session
	⟨200⟩ Response Body: factor_id*(uuid)*		
Namespace: /api/passwords			
GET	`/check/hash/<prefix>` Request Path: prefix*(str)* ⟨200⟩ Response Body: list of pw_suffix*(bytes)* ⟨200⟩ Response Header: `X-Realsize-Bytes`	Check if Password is weak	
Namespace: /api/sessions			
DELETE	/ ⟨204⟩ Empty Response	Logout This Session	×
GET	/ ⟨200⟩ Response Body: list of: created_at*(epoch)*, expires_at*(epoch)*, session_id*(uuid)*, user_agent_info*(dict)*	Get User Sessions	×
PATCH	/ Request Body: refresh_token*(str)* ⟨200⟩ Response Body: access_token*(str)*, refresh_token*(str)*	Refresh This Session	expired
DELETE	`/<session_id>` Request Path: session_id*(uuid)* ⟨204⟩ Empty Response	Logout A User Session	×

D Digitaler Anhang

Auf dem beigefügten Datenträger befindet sich eine Ordnerstruktur mit den folgenden Daten:

```
/
├── literature/ ........................ Verwendete Literaturquellen
├── src/ ................................ Quellcode der Applikation
│   ├── client/ ........................ Client-Dateien der Applikation
│   ├── server/ ........................ Server-Dateien der Applikation
│   ├── tests/ ................... Quellcode für die durchgeführten Tests
│   │   └── README.md ............... Anleitung zur Ausführung der Tests
│   └── README.md ........... Anleitung zur Ausführung der Anwendung
└── thesis.pdf ..................... Die Master-Thesis als PDF-Datei
```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen der Arbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Ich erkläre ferner, dass ich die vorliegende Arbeit in keinem anderen Prüfungsverfahren als Prüfungsarbeit eingereicht habe oder einreichen werde.

Die eingereichte schriftliche Fassung entspricht der auf dem Medium gespeicherten Fassung.

Ort, Datum	Unterschrift