HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

FIDO2-Erweiterung von TLS 1.3 in einer gängigen Kryptobibliothek

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

eingereicht von:	Mario Freund
geboren am:	29.05.2000
geboren in:	Berlin
Gutachter/innen:	Prof. Dr. Jens-Peter Redlich
	Prof. Dr. Ernst-Günter Giessmann

eingereicht am: ..... verteidigt am: .....

Zusammenfassung

Im 21. Jahrhundert, in dem es mittlerweile in Deutschland mehr als 60 Millionen Internetnutzer gibt [5], scheint eine Authentifizierung per Passwort nicht mehr zeitgemäß. Mehrfachverwendungen und unzureichende Sicherheitsstandards können von Angreifern ausgenutzt werden, um sich Zugang zu IT-Systemen zu verschaffen.
Anfang 2020 reichte Tom-Lukas Johann Breitkopf zu diesem Thema eine Bachelorarbeit ein, die sich mit der Kombination des Protokolls zur sicheren Datenübertragung TLS in der Version 1.3 und der Spezifikation FIDO2 zur passwortlosen Authentifizierung befasste [1]. Er entwickelte eine Spezifikation, die FIDO2 TLS 1.3 als neue Authentifizierungsmethode hinzufügt (TFE-Handshake) und implementierte einen Proof of Concept (PoC) in einer eher weniger verbreiteten Bibliothek.

Diese Arbeit greift die von ihm erarbeitete Spezifikation auf und entwickelt eine neue PoC-Implementierung, die in einer verbreiteten Kryptobibliothek erfolgt. Sie wird anhand von festgelegten Kriterien ausgewählt. Dabei geht es vor allem darum, die Erweiterung mit FIDO2 der Praxis näher zu bringen.
Es wird im Laufe der Arbeit gezeigt, dass besagte Implementierung auch in einer systemnahen Sprache möglich ist und dass eine Erweiterung auch praktisch umsetzbar ist, wenn WebAuthn-Server und TLS-Server als separate Prozesse laufen.
Des Weiteren werden durch Breitkopf bereits angesprochene Themen aufgegriffen und vertieft. Beispielsweise wird ein Mechanismus implementiert, der den Registrierungsstatus verschleiert, wenn kein Nutzer unter dem angegebenen Nutzernamen registriert ist. Dazu wird am Ende ein konkreter Vorschlag gemacht, wie man auch die Registrierung im TFE-Handshake unterbringen kann.

Insgesamt wird die Erweiterung mit FIDO2 auf ein neues Niveau gebracht und damit ein weiterer Schritt in Richtung der Verbreitung passwortloser Authentifizierungen getan.

Inhaltsverzeichnis

Abbildungsverzeichnis und Tabellenverzeichnis

Abbildungen

Tabellen

Abkürzungsverzeichnis

AAGUID	Authenticator Attestation Globally Unique Identifier
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
BSD	Berkeley Software Distribution
CTAP	Client to Authenticator Protocol
DB	Datenbank
DNS	Domain Name System
DoS	Denial of Service
DTLS	Datagram Transport Layer Security
(EC)DHE	(Elliptic Curve) Diffie-Hellman Exchange
ECDSA	Elliptic Curve Digital Signature Algorithm
FIDO(2)	Fast Identity Online (Version 2)
FI	FIDO2 with ID
FN	FIDO2 with Name
GPL	General Public License
GPLv2+	General Public License Version 2 or later
GPLv3	General Public License Version 3
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
IP	Internet Protocol
JSON	JavaScript Object Notation
LGPLv2.1+	Lesser General Public License Version 2.1 or later
LGPLv3+	Lesser General Public License Version 3 or later
MAC	Message Authentication Code

MIT	Massachusetts Institute of Technology
PIN	Persönliche Identifikationsnummer
PKCS	Public Key Credential Source
PoC	Proof of Concept
PSK	Preshared Key
RFC	Request for Comments
RP	Relying Party
RTT	Round Trip Time
SMPTS	Simple Mail Transfer Protocol Secure
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transport Control Protocol
TFE	TLS 1.3 with FIDO2 Extension
TLS	Transport Layer Security
U2F	Universal 2nd Factor
URL	Uniform Resource Locator
VPN	Virtual Private Network
WebAuthn	Web Authentication

1 Einleitung

Mit etwa 62,9 Millionen Menschen gibt es in Deutschland mehr Internetnutzer[1] als je zuvor [5]. Dazu werden auch immer mehr verschiedene Online-Dienste verwendet wie Videostreaming, Shopping-Seiten, E-Mail-Clients, Social Media und anderes.
Um die Benutzerkonten vor unautorisierten Zugriffen und Missbrauch zu schützen, sind immer noch Passwörter die am meisten verbreitete Vorkehrung. Dabei gibt es mit deren Nutzung verschiedene Sicherheitsprobleme. Angefangen mit der Erstellung eines Passworts, sind viele von ihnen zu kurz, nicht variabel genug oder enthalten private Details der Nutzer wie Geburtsdatum und Namen. Das macht es Angreifern besonders leicht, Passwörter zu cracken, denn der kryptografische Schlüsselraum ist in einem solchen Fall sehr begrenzt. Eine Umfrage zum Tag der Passwortsicherheit 2019 durch Web.de ergab, dass 59% der befragten deutschen Internetnutzer ihre Passwörter für mehrere verschiedene Dienste verwenden, wobei 32% der Befragten bei 16 oder mehr Online-Diensten registriert sind [2], was eine große Angriffsfläche schafft. Des Weiteren verwendet über ein Drittel der Befragten keine Sonderzeichen in ihren Passwörtern [2]. In einigen Fällen kommt erschwerend hinzu, dass Anbieter von Online-Diensten Sicherheitsvorgaben für Passwörter nicht oder nur unzureichend einfordern, um eine höhere Nutzerfreundlichkeit zu schaffen [6].
Eine Möglichkeit, um eine höhere Nutzerakzeptanz zu erzielen und Sicherheitsempfehlungen umzusetzen, z.B. die des BSI [3], sind Passwortmanager, welche kryptografisch hochwertige Passwörter generieren können und bei hinreichender Verschlüsselung auch sicher verwahren. 2016 benutzten etwa ein Drittel der deutschen Internetnutzer einen Passwortmanager [4].

Allerdings gibt es mittlerweile Alternativen, die in ihrer Umsetzung noch nutzerfreundlicher und gegen verschiedene Angriffsformen resistent sind. Um einen solchen Standard zu erarbeiten, hat sich 2012 die FIDO Alliance gegründet, deren Ziel die Etablierung passwortloser Authentifizierungen ist [7]. Mittlerweile gehören ihr viele große Konzerne wie Google, Amazon und Microsoft an [8].
Die Idee ist es, den Nutzer durch ein sogenanntes Challenge-Response-Verfahren zu authentifizieren. Möchte Bob Alice authentifizieren, sendet er eine „Challenge“ an Alice, sozusagen eine Aufgabe, die gelöst werden muss. Alice berechnet mithilfe der Challenge und mit einem Geheimnis, welches nur sie kennt, eine „Response“ und sendet diese zurück an Bob. Nun kann Bob die Response von Alice verifizieren und sie damit authentifizieren [9]. Der genaue Ablauf eines solchen Verfahrens wird am Beispiel von FIDO2 in Abschnitt 4 gezeigt.

Die nachfolgende Arbeit soll sich mit einer Kombination der beiden Standards FIDO2 und dem de-facto-Standard für sichere Datenübertragung TLS in der Version 1.3 befassen. Dabei soll die clientseitige Authentifizierung mit FIDO2 TLS 1.3 hinzugefügt werden, um eine nutzerfreundliche Alternative zur Verwendung von Clientzertifikaten zu schaffen, welche nach wie vor dominiert [10]. Die Authentifizierung mit FIDO2 könnte so weiterverbreitet und gleichzeitig die Clientauthentifizierung komfortabler werden.
Tom-Lukas Johann Breitkopf hat zu diesem Thema bereits eine Bachelorarbeit verfasst und einen Standard für einen TLS-Handshake mit FIDO2-Authentifizierung erarbeitet [1]. Ziel dieser Arbeit ist es, die Spezifikation von Breitkopf zu verwenden, um eine neue PoC-Implementation in einer verbreiteten Kryptobibliothek zu erarbeiten. Sie soll einen weiteren Beitrag zur Umsetzung dieser Spezifikation leisten.
Dafür werden als erstes TLS 1.3 und FIDO2 im theoretischen Teil genauer erläutert.

[1]In dieser Arbeit wird das verallgemeinernde Maskulinum verwendet. In diesem Fall sind grundsätzliche alle Geschlechter gemeint.

Anschließend wird die Spezifikation von Breitkopf vorgestellt und die Sinnhaftigkeit sowie Vor- und Nachteile einer solchen Erweiterung erläutert. Danach wird anhand sinnvoller Kriterien eine Bibliothek ausgewählt, in der die Implementation umgesetzt werden soll. Im praktischen Teil werden Anforderungen an die Implementierung formuliert. Sodann werden zusätzliche Hilfsmittel und Werkzeuge für die Implementierung bestimmt. Anschließend werden ausgewählte Aspekte der Implementierung beleuchtet, zu denen Einschränkungen und Designentscheidungen gehören. Zum Schluss wird die Anforderungserfüllung ausgewertet und es soll ein besonderer Blick auf die Performance geworfen werden.

2 Verwandte Arbeiten

Anfang des Jahres 2020 beschäftigte sich bereits Tom-Lukas Johann Breitkopf mit einer TLS-Erweiterung mit FIDO2 in seiner Bachelorarbeit „FIDO2 als TLS-1.3-Erweiterung“ [1].
Neben einer kompletten Spezifikation, die auch diese Arbeit verwendet, implementierte er außerdem einen PoC in der Bibiothek tlslite-ng. Diese Bibliothek ist in Python geschrieben und ermöglicht eine unkomplizierte und vergleichsweise schnelle Implementierung. Allerdings werden in dieser Bibliothek keine Serverzertifikate validiert, weshalb die Endpunktauthentifizierung nicht gewährleistet und die Bibliothek deshalb nicht sehr verbreitet ist [1]. Andere kleinere Schwachstellen in diesem PoC führen nicht zwingend zu Sicherheitsproblemen [1].
Für die Spezifikation diskutierte Breitkopf verschiedene Varianten und bewertete diese Anhand von Performance und Sicherheitsaspekten. Heraus kam der TFE (TLS 1.3 with FIDO2 Extension) – Handshake [1], der in dieser Arbeit ebenfalls verwendet wird. Er wird in Abschnitt 5 genau vorgestellt.

3 Transport Layer Security

Transport Layer Security, kurz TLS, ist ein 1994 eingeführter Standard zur sicheren Datenübertragung im Internet. Die erste Version wurde von Netscape Communications entwickelt und hieß damals noch Secure Socket Layer (SSL) 1.0 [11].
Mittlerweile ist TLS 1.3 die aktuelle Version des Protokolls aus dem Jahr 2018 [10]. Die folgende Abbildung soll die Position von TLS im TCP/IP-Referenzmodell darstellen.

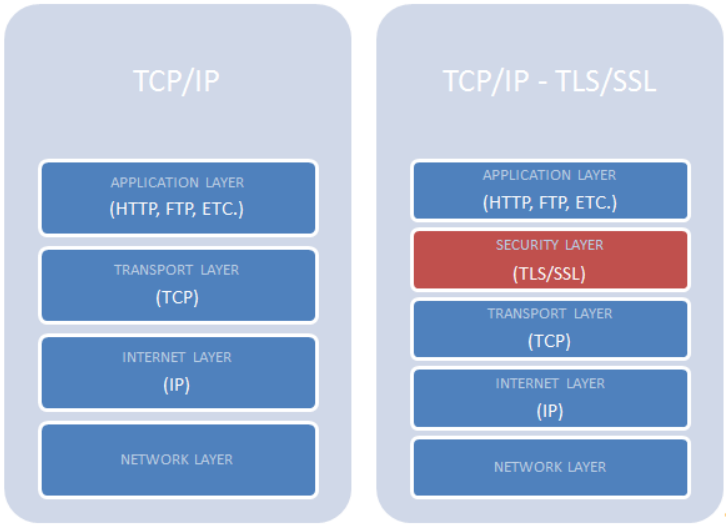

Abbildung 3.1: TLS im TCP/IP- Referenzmodell
Quelle: [13]

Im Protokollstapel setzt TLS also direkt auf TCP auf und verlässt sich dabei auf dessen Zuverlässigkeitseigenschaft [10]. Programmierer können TLS daher wie einen Socket benutzen [12].
Das Protokoll besteht im Wesentlichen aus zwei Hauptkomponenten: Dem Handshake- und dem Record-Protokoll. Dazu kommen das kleinere Alert- und in früheren Versionen das Change Cipher Spec-Protokoll [10].
Als Erstes werden die genauen Schutzziele des Protokolls betrachtet und anschließend das für diese Arbeit signifikante Handshake-Protokoll vorgestellt.

3.1 Schutzziele

TLS bedient die drei Schutzziele Authentifizierung, Vertraulichkeit und Integrität [10].

Authentifizierung

Im Kontext von TLS geht es um die Endpunktauthentifizierung der Sitzungsteilnehmer Client und Server [14]. Eine Authentifizierung ist für den Server in praktischen Umgebungen ein Muss, aber auch die Authentifizierung des Clients kann durch den Server gefordert werden [10].
Unter Authentifizierung versteht man die Überprüfung, ob die angegebene Identität eines Subjekts (in diesem Fall Endpunkts) mit der tatsächlichen Identität übereinstimmt [14]. Bisher geschieht das in TLS über Zertifikate (asymmetrische Kryptografie) oder einen symmetrischen PSK (pre-shared key) [10]. Die Erweiterung mit FIDO2 soll dem eine weitere Möglichkeit für die Clientauthentifizierung hinzufügen.

Vertraulichkeit

TLS stellt die Vertraulichkeit von Informationen sicher [10]. Das heißt, dass keine unautorisierte Informationsgewinnung stattfinden darf, was durch Verschlüsselung der zu transportierenden Daten sichergestellt wird, sodass kein Dritter diese Daten mitlesen kann [14].

Integrität

Es ist außerdem erforderlich, dass versendete Daten durch Dritte nicht verändert werden können, ohne dass es ein Sitzungsteilnehmer merkt [10].
Diese Aufgabe nimmt innerhalb von TLS das Record-Protokoll wahr, denn versendete TLS-Fragmente werden mit einem Message Authentication Code (MAC) abgesichert.

Der Rest des Abschnitts 3 bezieht sich auf RFC 8446 [10], der die Standardisierung des Protokolls enthält.

3.2 Handshake-Protokoll

Das Handshake-Protokoll wird zu Beginn einer jeden Sitzung verwendet, um verschiedene Parameter der Sitzung zwischen Client und Server auszuhandeln. Dazu gehören die Auswahl einer Protokollversion, einer Cipher Suite, die ggf. gegenseitige Authentifizierung und das Aushandeln von Schlüsselmaterial.

3.2.1 Ablauf

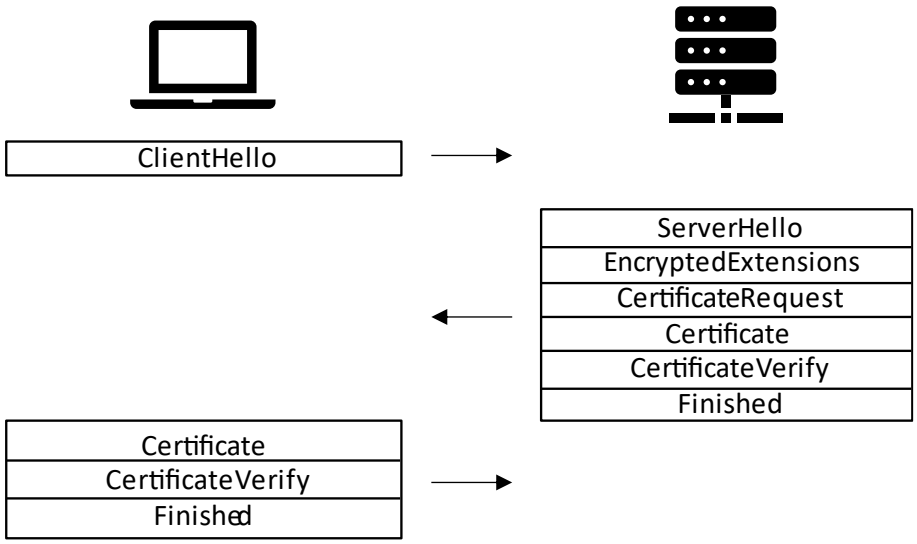

Abbildung 3.2.1: TLS 1.3-Handshake mit (EC)DHE, mit Client-Authentifizierung
Quelle: Eigene Abbildung nach [10, 15]

Der Handshake beginnt mit der **Erzeugung ephemerer Schlüsselpaare.** In der Client Hello-Nachricht sendet der Client unter anderem die von ihm unterstützten Protokollversionen, eine Liste von unterstützten Cipher Suites, Parameter zum Schlüsselaustausch und Client Hello-Erweiterungen. Auch die Erweiterung mit FIDO2 wird diesen Erweiterungsmechanismus nutzen, um dem Server die Verwendung von FIDO2 anzuzeigen. Für den Schlüsselaustausch

gibt es in TLS 1.3 drei wesentliche Modi. Wird ein PSK genutzt, ist die Nutzung von diesem schon im Client Hello angezeigt. Ansonsten wird ein (EC)DHE (Elliptic Curve Diffie-Hellman Exchange) verwendet. Allerdings kann auch mit einem PSK ein neuer Schlüssel über einen (EC)DHE erzeugt werden. Wird auf diese Weise Schlüsselmaterial erzeugt, so ist die Verbindung „forward secure“. Eine Kompromittierung eines Langzeitschlüssels hat also in diesem Fall keine Auswirkungen auf die Vertraulichkeit. Die notwendigen Parameter werden im Feld „key_share“ übertragen.
Der Server empfängt und verarbeitet das Client Hello. Dazu wählt er die zu verwendende Version und eine Cipher Suite aus. Gibt es keine Version oder Cipher Suite, die beide Endpunkte unterstützen, wird mit einem entsprechenden Alert abgebrochen. Auch aus den (EC)DHE-Gruppen des Feldes „supported_groups“ muss eine passende Gruppe auswählbar sein. Anschließend berechnet der Server das Schlüsselmaterial anhand der durch den Client zur Verfügung gestellten Parameter und sendet das Server Hello, unter anderem mit der ausgewählten Cipher Suite und den entsprechenden (EC)DHE-Informationen.
Eine sehr wichtige Neuerung in TLS 1.3 ist, dass alle Nachrichten nach dem Server Hello bereits verschlüsselt sind. Auch die Erweiterung mit FIDO2 kann von diesem Umstand profitieren.

Nach der Schlüsselerzeugung teilt der Server die **Serverparameter** mit. Das geschieht zum einen in der auf das Server Hello folgenden Nachricht Encrypted Extensions. Dort antwortet der Server ggf. auf die Erweiterungen aus dem Client Hello. Zum anderen kann der Server in einer Certificate Request-Nachricht ein Client-Zertifikat fordern, welches zur Authentifizierung des Clients herangezogen wird.

Die **Authentifizierung** ist die letzte Phase des Handshakes. Wird kein PSK verwendet, authentifiziert sich der Server über ein Server-Zertifikat, welches er in der Certificate-Nachricht sendet.
Um den öffentlichen Schlüssel zu verifizieren, erhält der Client auch eine Certificate Verify-Nachricht durch den Server. Sie enthält eine Signatur über den bisherigen Handshake mit dem privaten Schlüssel des Servers. Das Signaturverfahren muss einem entsprechen, welches der Client im Client Hello im Feld „signature_algorithms“ angeben hat. Nun weiß der Client, dass der Server im Besitz des zugehörigen privaten Schlüssels ist.
Zuletzt wird in der Finished-Nachricht mit Hilfe der ausgehandelten symmetrischen MAC-Schlüssel ein MAC über den gesamten Handshake berechnet, was dessen Integrität sicherstellt. Außerdem werden so die Identitäten der Endpunkte an die ausgehandelten Schlüssel gebunden. Bei der Verwendung eines PSK wird mit der Finished-Nachricht der Handshake authentifiziert. Fordert der Server ein Client-Zertifikat, sendet auch der Client die Certificate- und Certificate Verify-Nachricht.

Der grundlegende Ablauf eines Handshakes mit dem Austausch von (EC)DHE-Parametern und mit Client-Authentifizierung ist der Abbildung 3.2.1 zu entnehmen. Soll der Client sich authentifizieren, sendet der Server zusätzlich eine Certificate Request-Nachricht und der Client antwortet entsprechend ebenfalls mit Certificate und Certificate Verify.
Wird ein PSK verwendet, fallen die Nachrichten bezüglich der Zertifikate auf beiden Seiten weg, da die Authentifizierung implizit über den PSK erfolgt.

3.2.2 Erweiterungen

TLS bietet die Möglichkeit, über das Client Hello Extensions an den Server zu schicken. Diese werden über eine ID eindeutig identifiziert und können zusätzlich auch weitere Daten beinhalten. In einigen Fällen reicht es aber, wenn die Verwendung dieser Erweiterung nur über die ID angezeigt wird, ohne weitere Daten zu senden.
Der Server kann auf die Client Hello Extensions in den Nachrichten Server Hello, Encrypted Extensions, Hello Retry Request oder Certificate antworten.
Auch der Server kann per Certificate Request-Nachricht an den Client Erweiterungen anfragen. Der Client kann in seiner Certificate-Nachricht darauf antworten.
Für jede Erweiterung muss des Weiteren klar definiert sein, in welcher Nachricht sie vorkommen kann.

Eine Liste mit IANA-registrierten Erweiterungen ist dem RFC 8446 zu entnehmen.

3.3 Weitere Bestandteile

Neben dem Handshake-Protokoll besteht TLS aus dem Record-Protokoll, welches für das Versenden von TLS-Nachrichten und die Integritätssicherung dieser zuständig ist und dem Alert-Protokoll, welches Alerts definiert, die versendet werden, wenn es zu Fehlern kommt oder die Verbindung beendet werden soll.
Das Alert-Protokoll definiert festgelegte Alert-Typen. Ein Alert kann ein fataler Alert, der immer zur Beendung der Verbindung führt, oder eine Warnung sein. Entscheidend ist, dass durch Alerts keine inneren Systeminformationen preisgegeben werden, durch die die Schutzziele des Protokolls beeinträchtigt werden könnten. Die Erweiterung mit FIDO2 wird drei zusätzliche Alerts einführen.

Auf das Record-Protokoll soll hier nicht weiter eingegangen werden, denn es wird im Rahmen dieser Arbeit nicht ergänzt oder verändert.

4 FIDO2

FIDO2 definiert die beiden Teilspezifikationen WebAuthn (Web Authentication Specification) und CTAP (Client-to-Authenticator Protocol) [18].
Die in der WebAuthn-Spezifikation definierte WebAuthn API ermöglicht es Online-Diensten, über den Browser des Clients die Authentifizierung mit FIDO2 durchzuführen. Das CTAP wiederum definiert eine Schnittstellte zwischen dem Client und einem externen Authentifikator, um kryptografische Operationen zur Authentifizierung durchzuführen [18]. Ein Authentifikator kann in verschiedenen Formen vorliegen, beispielsweise extern als USB-Token, Smartphone des Clients oder intern als Software-Applikation [20].
In dieser Arbeit wird meist von einem externen Authentifikator in Form eines USB-Tokens ausgegangen. Dazu wird im Kontext von FIDO2 von „Relying Party“ (RP) anstelle von „Online-Dienst“ gesprochen.
Abbildung 4.1 visualisiert den schematischen Aufbau noch einmal.

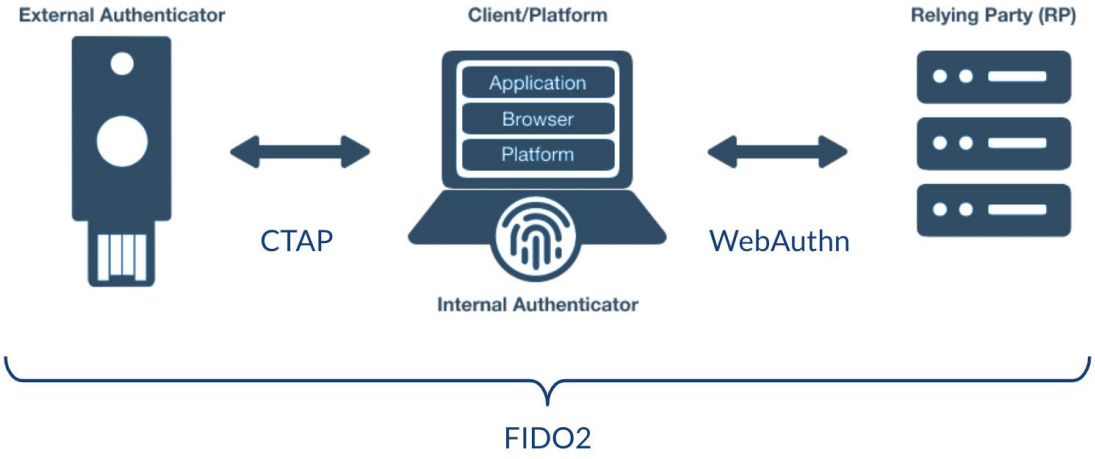

Abbildung 4.1: FIDO2
Quelle: [20]

Zur Registrierung bei einem Online-Dienst wird dafür auf dem Token des Clients ein neues Schlüsselpaar, bestehend aus öffentlichem und privatem Schlüssel, generiert. Der öffentliche Schlüssel wird anschließend beim Online-Dienst hinterlegt und der private Schlüssel sicher auf dem Token verwahrt [19].
Bei der Anmeldung sendet der Online-Dienst eine zufällig generierte Challenge an den Client. Dieser signiert diese mit seinem privaten Schlüssel, wofür er mit dem Token interagieren muss, was meist mit einer PIN-Eingabe oder einem Fingerabdruck geschieht. Die signierte Challenge wird als Response an den Online-Dienst zurückgeschickt, der die Response mit dem öffentlichen Schlüssel verifiziert. Bei Erfolg ist der Client authentifiziert [19].

Wie bei der Verwendung von Passwörtern handelt es sich hier also auch um Authentifizierung durch Wissen [14]. Allerdings lässt sich schnell erkennen, dass der Client nicht nur über das Wissen verfügen muss, um die richtige Response zu erzeugen, sondern es handelt sich hier auch um Authentifizierung durch Besitz, da der Token im Besitz des Nutzers sein muss. Wird das Verfahren zusätzlich mit einem biometrischen Verfahren (Authentifizierung durch Sein [14]) kombiniert, also z.B. mittels Fingerabdruck, gibt es sogar drei Dimensionen der Authentifizierung.

In FIDO2 ist der alte Standard U2F als Unterspezifikation mit enthalten, was eine Abwärtskompatibilität ermöglicht [18].

4.1 Sicherheitsbetrachtungen

Mit FIDO2 gehen einige Sicherheitsbetrachtungen einher. Insgesamt sind im Dokument „FIDO Security Reference“ 16 Sicherheitsziele des Standards aufgelistet [21].
Zu den wichtigsten gehören eine starke Authentifizierung, d.h. eine hohe kryptografische Sicherheit, die Minimierung der Preisgabe von internen Informationen an die RP und die Resistenz gegen verschiedene Angriffsformen.
Zu den Angriffsformen zählen Denial-of-Service (DoS)- Attacken, Replay-Attacken, Impersonation-Attacken und Parallel Session-Attacken.
Des Weiteren soll die ungewollte Preisgabe von Informationen durch die RP oder den Authentifikator nicht dazu führen, dass ein Angreifer sich als ein anderer Benutzer ausgeben kann.
Die RPs müssen auch kryptografisch gesehen voneinander getrennt sein, sodass keine RP Registrierungs- oder Authentifizierungsinformationen über eine andere RP erhalten kann.
Es soll außerdem möglichst schwierig sein, Credentials eines Nutzers zu erraten, um sich als ihn auszugeben und somit die Authentifizierung zu untergraben [21].

Um diese Ziele einzuhalten, sind im FIDO-Standard verschiedene Sicherheitsmaßnahmen vorgesehen.
Der Schlüssel zur Authentifizierung ist sicher auf dem Authentifikator verwahrt und kann bei entsprechender Konfiguration nur durch Nutzerverifizierung zur Nutzung freigegeben werden. Diese Maßnahme ist entscheidend für zahlreiche Sicherheitsziele, vor allem für eine starke Authentifizierung und die Verhinderung vom Erraten des Schlüssels, da der private Schlüssel den Authentifikator niemals verlässt [21].
Der Schlüssel ist außerdem einzigartig für die Kombination aus RP, Benutzer und Authentifikator. Er ist dadurch auf diesen Nutzungsbereich eingeschränkt und wird für keine andere RP verwendet, was die Trennung der RPs ermöglicht und gleichzeitig die Preisgabe an Informationen minimiert. Darüber hinaus kann ein Angreifer von einem Datenleck bei einer RP insofern nicht profitieren, als dass keine Informationen über andere RPs gewonnen werden können. Auch der zugehörige öffentliche Schlüssel nützt ihm nichts [21].
Um die Kopie eines Authentifikators aufzudecken, wurde ein Signaturzähler eingeführt, der bei jeder Signatur inkrementiert wird und auch beim Server hinterlegt ist. So kann der Server die Echtheit des Authentifikators auf Plausibilität überprüfen [21].
Um gegen verschiedene Netzwerk-Attacken gesichert zu sein, werden die Daten zur Registrierung und Authentifizierung nur über einen sicheren TLS-Kanal übertragen [21] und die RP-ID, die die RP identifiziert und bei Registrierung und Authentifizierung übermitteltet wird, muss einem registrierbaren Domain-Suffix der RP-Domain entsprechen. So sollen Man-in-the-Middle-Attacken verhindert werden [16]. Der „FIDO Security Reference“ [21] sind diverse weitere Schutzmaßnahmen zu entnehmen.

Ein weiterer interessanter Aspekt ist die Resistenz gegen Phishing-Attacken. Die nach wie vor sehr große Gefahr bei der Authentifizierung mittels Passwort ist, dass Nutzer auf falsche Webseiten umgeleitet werden und dort ihre Passwörter preisgeben. Dem Angreifer wäre es somit möglich, sich als dieser Nutzer auszugeben.

Wird der Nutzer während einer FIDO2-Authentifizierung auf eine Phishing-Seite umgeleitet, gibt es zwei denkbare Szenarien. Entweder es findet keine Authentifizierung statt, weil für diese RP kein privater Schlüssel auf dem Authentifikator ist oder es findet eine Authentifizierung mit einem zuvor bei der Registrierung auf dieser Seite erzeugten Schlüssel statt, was der RP der Phishing-Seite keine Informationen über andere RPs bringt [21].
Sollte es dem Angreifer gelingen, die PIN eines FIDO2-Tokens über einen Phishing-Angriff zu erhalten, bringt ihm auch das keinen nennenswerten Vorteil, da er nicht im Besitz des Tokens ist [21], den Fall ausgeschlossen, dass er als Man-in-the-Browser agiert und der Token eingesteckt ist.

4.2 Registrierung

Im nachfolgenden Teil soll die Registrierung mit Hilfe der Spezifikationen WebAuthn [16] und CTAP [22] genauer beschrieben werden. Für die Erweiterung von TLS 1.3 wird die Registrierung zwar nicht betrachtet. Es ist jedoch zum einen nötig, die Registrierung zu verstehen, um die Anmeldung nachzuvollziehen und zum anderen ist als weiterführendes Thema auch die Einbindung der Registrierung in TLS denkbar.
Die Erläuterung der Registrierung orientiert sich an Abbildung 4.2.

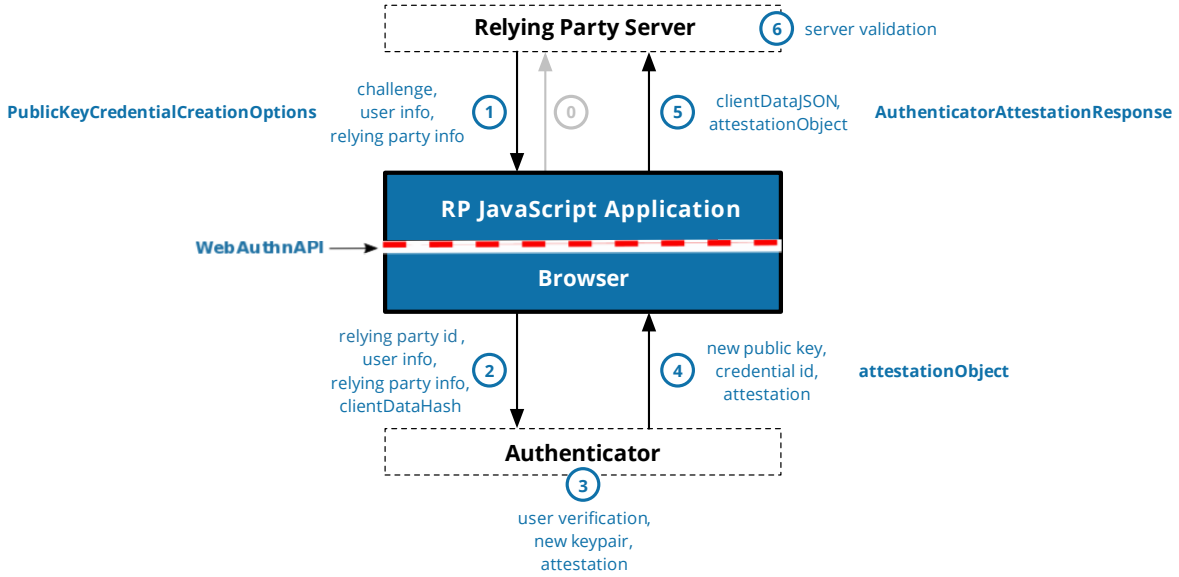

Abbildung 4.2: Ablauf der Registrierung
Quelle: [16]

Die Registrierung beginnt mit **Schritt 0,** bei dem der Client eine Registrierung bei der RP anfragt [17].
Daraufhin erzeugt die RP ein neues Dictionary „PublicKeyCredentialCreationOptions“, welches einige Parameter enthält [17]. Dazu gehören mindestens eine RP-ID, eine Nutzer-ID, unter der sich der Client registrieren möchte, eine Challenge, die aus zufällig generierten Bytes besteht, Typ und Algorithmus für die Generierung von neuen Public Key-Credentials und optional weitere Parameter, wie Timeout oder Extensions [16]. Alle Parameter können unter 5.4 der Spezifikation [16] nachgelesen werden. Die RP-ID dient dazu, die RP eindeutig zu identifizieren und entspricht standardmäßig der RP-Domain. Allerdings kann auch eine andere RP-ID übermittelt werden, die dann aber einem registrierbaren Domain-Suffix entsprechen muss. Dies hat der Client entsprechend zu prüfen [16]. Diese Daten werden in **Schritt 1** an die JavaScript-Applikation des Clients gesendet.

Darauf folgend in **Schritt 2** nutzt die JavaScript-Applikation die `navigator.credentials.create()`-Methode, um neue Credentials auf dem Authentifikator zu erzeugen [17]. Der Browser muss nun, wie oben angegeben, die RP-ID überprüfen [16] und nutzt dann die `authenticatorMakeCredential()`-Methode. Dafür muss er als erstes die „clientData“ bilden, ein Dictionary bestehend aus dem Typ der Operation, der hier auf „webauthn.create“ gesetzt werden muss, der base64url-codierten Challenge und der Web-Origin des Objekts, welches die WebAuthn-API zur Erzeugung der neuen Credentials aufgerufen hat [16]. Die „clientData“ werden JSON-serialisiert und anschließend wird ein SHA-256-Hash darüber gebildet. Die `authenticatorMakeCredential()`-Methode erhält als Argument diesen „clientDataHash“, Informationen über die RP, den Nutzer und die Informationen aus den „PublicKeyCredentialCreationOptions“ betreffend der Erzeugung einer neuen Public Key Credential Source (PKCS). Eine vollständige Liste der Argumente kann der Spezifikation [22] unter 5.1 entnommen werden.

In **Schritt 3** erzeugt der Authentifikator eine neue PKCS. Den zu verwendenden Algorithmus entnimmt er den „pubKeyCredParams“ aus den „PublicKeyCredentialCreationOptions“. Zuvor muss das Einverständnis des Nutzers eingeholt werden („user verification“) [22].
Für die Erzeugung einer PKCS gibt es zwei Möglichkeiten: Es kann ein „resident“ Schlüsselpaar erzeugt werden, welches im persistenten Speicher des Authentifikators hinterlegt wird oder es kann der private Schlüssel so vom Authentifikator verschlüsselt werden, dass nur er ihn wieder entschlüsseln kann [16]. Die erzeugte Chiffre wird dann bei der RP als ID hinterlegt. Auf diese Weise entsteht kein Speicherverbrauch für den Authentifikator. Allerdings wird dann zur Authentifizierung die ID der PKCS benötigt, wofür der Server den Nutzernamen benötigt, um die ID aus der Datenbank abzurufen [16]. Für die Erweiterung von TLS 1.3 wird das Probleme verursachen (siehe Abschnitt 5). Wird ein resident Key verwendet, reicht allein die RP-ID, um die richtige PKCS auszuwählen [16]. Eine PKCS wird eindeutig durch eine Credential-ID identifiziert und ist in ihrer Gültigkeit durch die RP-ID eingeschränkt [16] (siehe Abschnitt 4.1).

In **Schritt 4** muss der Authentifikator ein Attestation-Objekt an den Browser des Clients zurückliefern. Es beinhaltet die „authenticator data“, einen „attestation statement format identifier“ und ein „attestation statement“.
Die „authenticator data“ enthalten neben einem SHA-256-Hash über der RP-ID, Flags und einem Signaturzähler (siehe Abschnitt 4.1) insbesondere die „attested credential data“, die wiederum eine AAGUID zur Identifizierung des Authentifikator-Modells, die Credential-ID der erzeugten Credentials und den dazugehörigen öffentlichen Schlüssel enthalten [16].
Der „attestation statement format identifier“ enthält lediglich Formatinformationen über das „attestation statement“ [16, 22].
Das „attestation statement“ wiederum beinhaltet eine Signatur über die Konkatenation der „authenticator data“ und des „clientDataHashs“, also auch über die Challenge der RP mit dem privaten Attestierungsschlüssel des Authentifikators. Auch ein Attestierungszertifikat kann mitgeliefert werden. Es beinhaltet den öffentlichen Attestierungsschlüssel. Über die Attestierung werden bestimmte Eigenschaften des Authentifikators bestätigt [16].

Die JavaScript-Applikation, die das Attestation-Objekt als Rückgabewert der `navigator.credentials.create()`-Methode erhalten hat, liefert genanntes als „AuthenticatorAttestationResponse“ zusammen mit den zuvor generierten „clientDataJSON“ an den RP-Server zurück [16] (**Schritt 5**).

Zuletzt in **Schritt 6** validiert die RP die „AuthenticatorAttestationResponse“. Dazu werden die „clientDataJSON“ überprüft, die Signatur verifiziert und ggf. die Zertifikatskette des Attestierungszertifikat untersucht. Ist der Client inkl. seines Authentifikators als vertrauenswürdig einzustufen, wird die ID der PKCS und weitere Informationen über den Client in der Datenbank des Servers gespeichert. In diesem Fall war die Registrierung erfolgreich [16, 17].

4.3 Authentifizierung

Bei der Authentifizierung handelt es sich im Grunde um einen ähnlichen Ablauf wie bei der Registrierung. Die folgenden Erläuterungen zu Abbildung 4.3 bilden die Grundlage für die Erweiterung von TLS 1.3 mit FIDO2, da die dargestellten Schritte der Authentifizierung in den Handshake eingebunden werden müssen.

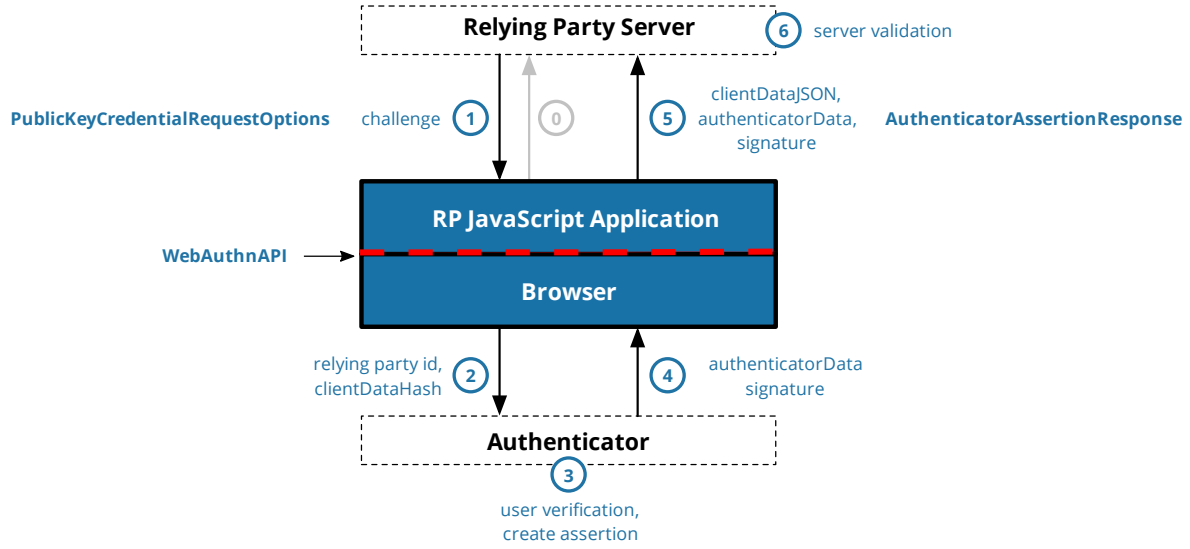

Abbildung 4.3: Ablauf der Authentifizierung
Quelle: [16]

Der Client beginnt mit **Schritt 0**, der initialen Anfrage [16]. Wie diese konkret aussieht, hängt vom Anwendungsfall ab. Die Erweiterung mit FIDO2 von TLS 1.3 wird diesen Schritt mit der entsprechenden Client Hello Extension durchführen.

Daraufhin erzeugt der RP-Server ein neues Dictionary „PublicKeyCredentialRequestOptions“, welches in **Schritt 1** an die JavaScript-Applikation des Clients gesendet wird [16]. Sie enthält die obligatorische Challenge, optional einen Timeout, der angibt, wie lange die RP bereit ist, auf eine Antwort zu warten, auch optional die RP-ID und eine Liste von Allow-Credentials, die die IDs von PKCSs enthält, falls die registrierten PKCSs nicht resident sind. Des Weiteren kann ein Wert „userVerification“ geliefert werden, der angibt, ob der Endnutzer im Zuge der Authentifizierung durch den Authentifikator verifiziert werden soll. Standardmäßig ist dieser Wert auf „preferred“ gesetzt. Zuletzt kann auch eine Liste von Extensions enthalten sein, die aber nicht Teil dieser Arbeit sein sollen [16]. Die Daten werden mit der `navigator.credentials.get()`-Methode an den Browser weitergegeben.

Schritt 2 besteht aus dem Ruf der `authenticatorGetAssertion()`-Methode durch den Browser. Dazu muss er, falls erhalten, die RP-ID daraufhin überprüfen, ob sie der Domain oder einem registrierbaren Domain-Suffix des aufrufenden Objekts entspricht. Falls die RP-ID nicht

geliefert worden sein sollte, wird sie auf die effektive Domain des aufrufenden Objekts gesetzt [16]. Auch bei der Authentifizierung muss der Browser nun die „clientData“ erzeugen. Der Typ wird in diesem Anwendungsfall auf „webauthn.get“ gesetzt. Die Challenge wird wieder base64url-codiert. Die Web-Origin des aufrufenden Objekts wird gesetzt und optional können Parameter eines Token Bindings enthalten sein (siehe [16] unter 5.1.4.1) [16]. Die „clientData“ werden JSON-serialisiert und ein SHA-256-Hash darüber gebildet.
Nun werden die RP-ID, der „clientDataHash“, optional die Liste mit Allow-Credentials, Extensions und Flags für „user presence“ (ob die Nutzeranwesenheit überprüft werden soll) und „user verification“ (ob der Nutzer verifiziert werden soll) an die `authenticatorGetAssertion()`-Methode übergeben [23].

In **Schritt 3** generiert der Authentifikator eine Assertion. Er muss dazu als Erstes alle PKCSs bestimmen, die an die übergebene RP-ID gebunden sind. Das kann er entweder über die Liste der Allow-Credentials tun oder, falls diese nicht vorhanden ist, im persistenten Speicher. Bevor weitere Operationen erfolgen, muss entsprechend der Flags der Nutzer verifiziert oder seine Anwesenheit festgestellt werden. Anschließend muss der Authentifikator eine PKCS auswählen. Bei einer Liste mit Allow-Credentials wird dazu eine beliebige passende ausgewählt. Sind mehrere resident PKCSs unter dieser RP-ID vorhanden, wird der Nutzer aufgefordert eine PKCS auszuwählen [22].
Nun wird mit Hilfe der ausgewählten PKCS auch hier eine Signatur über die „authenticatorData“ und den „clientDataHash“ gebildet. Die „authenticatorData“ bestehen erneut aus mindestens einem SHA-256-Hash über der RP-ID, Flags und dem Signaturzähler. Der genaue Aufbau kann unter 6.1 in der Spezifikation [16] nachgelesen werden.

In **Schritt 4** liefert der Authentifikator die „authenticatorData“ und die erzeugte Signatur zurück. Wenn nicht genau eine Credential-ID in der Liste der Allow-Credentials enthalten war, muss außerdem die Credential-ID der verwendeten PKCS mitgeliefert sein. Wurde eine resident PKCS verwendet, muss zusätzlich die User-ID des Nutzers beigefügt sein [22]. Eine genaue Rückgabeliste kann unter 5.2 der Spezifikation [22] nachgelesen werden.

In **Schritt 5** sendet die JavaScript-Applikation die über den Rückgabewert der `navigator.credentials.get()`-Methode erhaltenen Daten als „AuthenticatorAssertionResponse“ an den Server. Sie enthält die JSON-serialisierten „clientData“, die „authenticatorData“, die Signatur und weitere Parameter aus 5.2 der Spezifikation [22] [16].

Nun muss der RP-Server in **Schritt 6** die „AuthenticatorAssertionResponse“ validieren. Als erstes muss er überprüfen, ob, falls er bei Beginn der Authentifizierung eine Liste mit Allow-Credentials gesendet hat, die Credential-ID in der Antwort des Clients in dieser Liste vorhanden ist. Anschließend muss er den Nutzer identifizieren und in Erfahrung bringen, ob er tatsächlich im Besitz der entsprechenden PKCS ist. Neben anderen Schritten muss er dann die „authenticatorData“ und die JSON-serialisierten „clientData“ verifizieren. Des Weiteren muss er bestätigen, dass die erhaltene Signatur eine valide Signatur über den „authenticatorData“ und dem „clientDataHash“ ist. Zuletzt folgt eine Überprüfung des Signaturzählers. Er wird mit dem vom RP-Server gespeicherten Signaturzähler verglichen. Ist er größer, wird der der Signaturzähler beim RP-Server gespeichert. Ist er gleich oder kleiner, ist das ein Zeichen, dass der Authentifikator des Nutzers geklont worden sein könnte. Der Authentifikator sollte laut Standard für jede PKCS einen Signaturzähler verwahren. Allerdings

ist auch ein globaler Signaturzähler erlaubt [16]. Der RP-Server lässt das Ergebnis der Prüfung des Signaturzählers in eine Risikoberechnung einfließen. Ist die Risikoberechnung erfolgt und das Risiko akzeptabel, ist der Nutzer authentifiziert [16]. Die einzelnen Schritte sind unter 7.2 in der Spezifikation [16] zu finden.

5 Der TFE-Handshake

Der TFE-Handshake ist eine konkrete Realisierung der FIDO2-Erweiterung von TLS 1.3. Dabei muss zwischen der Client Hello Extension, die die Authentifizierung mit FIDO2 initiiert, und der FIDO2-Erweiterung als Gesamtes unterschieden werden. Denn es werden auch vier weitere Handshake-Nachrichten hinzugefügt.
Wie bereits am Anfang erwähnt, ist der Fokus dieser Arbeit auf die Authentifizierung und nicht auf die Registrierung gelegt. Diese kann bereits vorab z.B. über einen Webserver erfolgen, was auch eine bessere Nutzerinteraktion und Rückmeldung bei Fehlern in der einmaligen oder zumindest seltenen Registrierungsphase erlaubt, oder manuell in einer Datenbank des TLS-Servers. Die erfolgreiche Registrierung wird deshalb für den Handshake vorausgesetzt und spielt keine größere Rolle.
Bevor der modifizierte Handshake vorgestellt wird, soll vorher ein Blick auf die Vor- und Nachteile dieser Erweiterung geworfen werden.

5.1 Vorteile

Da es sich bei FIDO2 um ein deutlich sichereres Verfahren als die passwortbasierte Authentifizierung handelt, ist es denkbar, dass bei einer großen Verbreitung vollständig auf Passwörter verzichtet werden könnte [1]. Eine Erweiterung von TLS mit FIDO2 würde gerade eine solche Verbreitung begünstigen, da TLS von vielen Anwendungen verwendet wird, um eine sichere Datenübertragung zu ermöglichen.
Des Weiteren könnte FIDO2 so auf verschiedene andere TLS-basierte Anwendungsprotokolle als HTTPS portiert werden, beispielsweise auf VPN oder SMTPS [1]. Eine Integration von FIDO2 wäre bei einer entsprechenden Implementierung der Erweiterung für Anwender unkompliziert, da einige Konfigurationen ausreichen würden, um die Clientauthentifizierung mittels FIDO2 zu ermöglichen [1].
Ein Vorteil liegt zuletzt insbesondere auch in der Alternative zum Client-Zertifikat. Denn diese verursachen in der Praxis einige Probleme. Beispielsweise muss der Client die sichere Verwahrung eines solchen Zertifikats ermöglichen und eine entsprechende Anzahl an Zertifikaten verwalten. Das führt zu einer schlechten Nutzerfreundlichkeit [24]. Bei FIDO2 hingegen ist in der Regel lediglich ein kleiner Token zu verwahren, der auch am Schlüsselbund getragen werden kann. Für viele Nutzer wäre das vermutlich ein guter Kompromiss im Vergleich mit der unsichereren Passwortauthentifizierung.

5.2 Nachteile

Neben all den genannten Vorteilen gibt es auch Nachteile, die mit dieser Erweiterung einhergehen. Die Erweiterung mit FIDO2 könnte, wie theoretisch jede Erweiterung, neue Sicherheitslücken erzeugen, die zur Verletzung der Schutzziele führen. Im Falle von TLS hätte dies schwerwiegende Folgen, denn etliche Anwender sind auf eine sichere Datenübertragung angewiesen [1].
Außerdem könnte eine geringere Nutzerakzeptanz Probleme bereiten, denn die Authentifizierung bereits im TLS-Handshake würde dazu führen, dass die Nutzer sich authentifizieren müssten, bevor die eigentliche Webseite geladen wäre. Für fachfremde Nutzer könnte das sehr ungewohnt sein [1].
Ein weiterer möglicher Nachteil könnte die erhöhte Latenz und die damit verbundene Aufblähung des Protokolls sein. Allerdings hat Breitkopf in seiner Arbeit gezeigt, dass dies

nicht unbedingt der Fall sein muss [1]. Dieser Aspekt wird im Laufe dieser Arbeit (Abschnitt 7.4.2) erneut betrachtet und im Zuge der neuen Implementierung bewertet.

5.3 Ablauf des Handshakes

Aufgrund der zwei unterschiedlichen Arten während des Registrierungsprozesses eine neue PKCS zu erzeugen, muss auch der TFE-Handshake in zwei unterschiedlichen Varianten möglich sein. Handelt es sich um eine resident PKCS, kann der Server vom Nutzer unabhängige „PublicKeyCredentialRequestOptions“ senden und dann anhand der Nutzer-ID die richtige PKCS auswählen und die Response verifizieren.
Ist die PKCS nicht resident, benötigt er den Nutzernamen des Clients, um ihm die benötigten Allow-Credentials zukommen zu lassen. Allerdings soll es für einen Dritten nicht möglich sein, für einen Nutzernamen in Erfahrung zu bringen, ob dieser registriert ist oder nicht. Deshalb muss der Nutzername durch das Schlüsselmaterial geschützt an den Server gesendet werden [1].
Breitkopf diskutierte in seiner Arbeit verschiedene Möglichkeiten, um den TFE-Handshake umzusetzen. Kriterien waren dabei unter anderem der angesprochene Schutz des Registrierungsstatus, die Vermeidung von zusätzlichen RTTs und die Gewährleistung von „perfect forward secrecy“, also dass pro Sitzung frisches Schlüsselmaterial ausgehandelt wird, sodass die Kompromittierung eines Langzeitschlüssels nicht die Entschlüsselung von ausgetauschten Daten alter Sitzungen ermöglicht [1].
Als Ergebnis präsentierte er einen doppelten Handshake und entsprechend die zwei Modi FI (FIDO with ID) und FN (FIDO with name). Die Architektur des Handshakes orientiert sich an der Verwendung von Zertifikaten zur Authentifizierung. Breitkopf erarbeitete eine Muss-, Soll- und Kann-Vorschrift nach RFC 2119 [51] für beide Modi, die im Folgenden zusammengefasst werden.
Die genauen Formate der neuen Nachrichten sind den Anhängen von [1] zu entnehmen.

5.3.1 FI-Modus

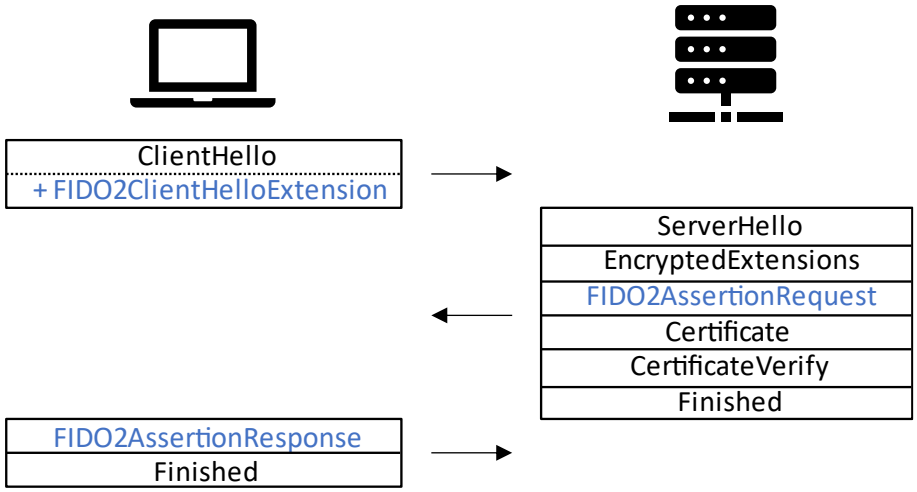

Abbildung 5.3.1: TFE-Handshake im FI-Modus
Quelle: Eigene Abbildung nach [1]

Der Ablauf im FI-Modus ist in Abbildung 5.3.1 dargestellt. Die neuen Nachrichten bzw. Erweiterungstypen sind blau markiert.

Der Client beginnt mit dem Client Hello und darin enthalten der FIDO2 Client Hello Extension, die die Authentifizierung mit FIDO2 anbietet. Die Extension enthält ein Feld für den Modus, welches auf FI gesetzt sein MUSS [1].
Der Server generiert mit Hilfe des Client Hello und den Parametern für das Server Hello das notwendige Schlüsselmaterial, um die Kommunikation nach dem Server Hello „forward secure“ zu verschlüsseln. Unterstützt er die FIDO2-Erweiterung nicht, so ignoriert er sie einfach. Hat der Client andersherum die FIDO2 Client Hello Extension nicht gesendet, KANN der Server mit einem Alert „fido2_required“ abbrechen [1].
In derselben RTT wie das Server Hello sendet der Server dann, falls beide Seiten die Erweiterung unterstützen, die FIDO2 Assertion Request-Nachricht. In ihr sind diverse Parameter der „PublicKeyCredentialRequestOptions“ [16] enthalten, die der Client benötigt, um eine Response zu erzeugen [1]. Die FIDO2 Assertion Request-Nachricht ist in diesem Fall nutzerunabhängig. Danach folgen die Certificate-, Certificate Verifiy-, und Finished-Nachrichten.
Mit den Informationen aus dem Server Hello erzeugt der Client nun seinerseits das Schlüsselmaterial und prüft die Certificate-, Certificate Verifiy-, und Finished-Nachrichten des Servers [1]. Er überprüft somit die Authentizität des Servers und die Integrität des bisherigen Handshakes.
Der Client MUSS nun die ggf. gesendete RP-ID des Servers wie in Abschnitt 4 beschrieben überprüfen. Entspricht die RP-ID nicht der Sever-Domain oder einem registrierbaren Domain-Suffix, MUSS der Client den Handshake mit einem „fido2_bad_request“-Alert abbrechen [1]. Er MUSS dann die „clientData“ erzeugen, ggf. Erweiterungen verarbeiten und entsprechend des Ablaufs der Authentifizierung die `authenticatorGetAssertion()`-Methode aufrufen [1]. Schlägt die Generierung einer Assertion fehl, MUSS der Client mit einem „internal_error“-Alert abbrechen oder, wenn der Nutzer die Aktion abgebrochen hat, mit einem „user_canceled“-Alert. Bei Erfolg erzeugt der Client eine FIDO2 Assertion Response-Nachricht, die die Rückgabewerte des Authentifikators zur Assertion enthält und die „clientData“. Es MUSS außerdem das Feld „user_handle“ mit der ID des Nutzers besetzt werden. Der Client sendet die Nachricht in derselben RTT wie die Finished-Nachricht [1].
Der Server kann den Nutzer anhand des „user_handles“ identifizieren und verifiziert nun, wie in Abschnitt 4 beschrieben, die Assertion. Beim Fehlschlag der Verifizierung MUSS der Server den Handshake mit einem „fido2_authentication_error“-Alert beenden. Bei Unstimmigkeiten, beispielsweise mit dem Signaturzähler, KANN der Server den Handshake mit angegebenen Alert beenden. Er KANN jedoch auch die Verbindung mit dem unauthentifizierten Client fortsetzen [1]. Die Anwendung könnte in diesem Fall dem Client nur bestimmte Rechte und Ressourcen zugänglich machen.
Hat der Server die Finished-Nachricht überprüft, ist der Handshake beendet und das Record-Protokoll kann genutzt werden, um Daten sicher auszutauschen.

5.3.2 FN-Modus

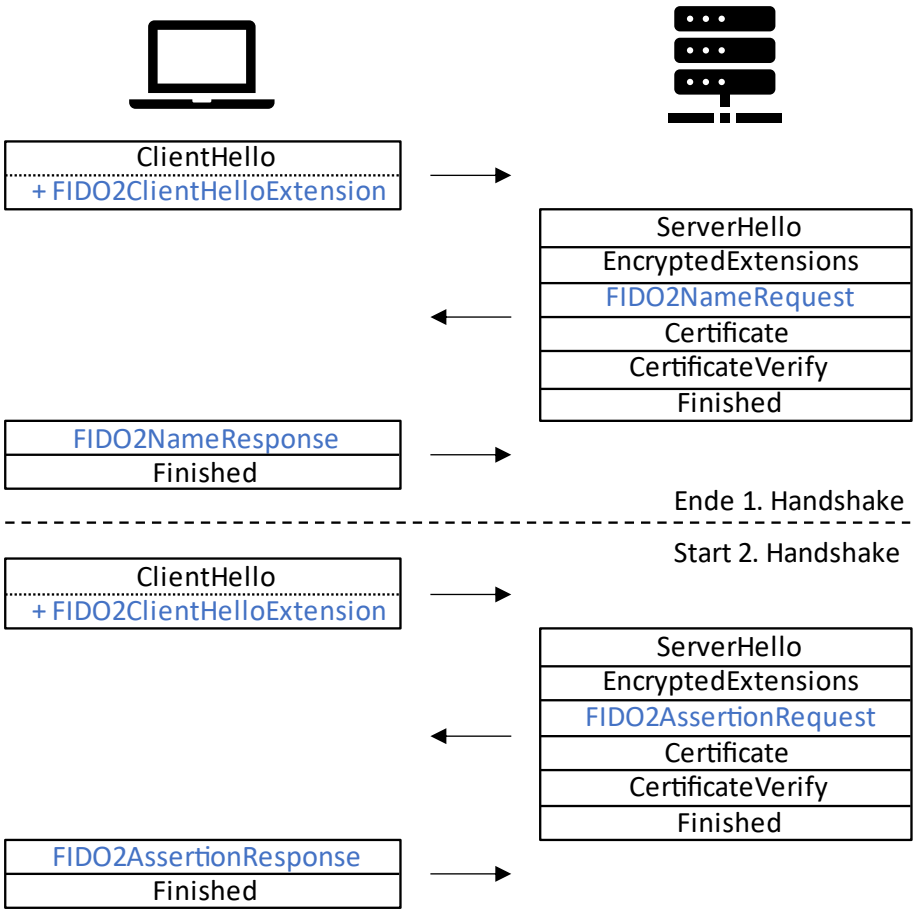

Abbildung 5.3.2: TFE-Handshake im FN-Modus
Quelle: Eigene Abbildung nach [1]

Im FN-Modus wird grundsätzlich der volle doppelte Handshake ausgeführt, wie in Abbildung 5.3.2 zu sehen. Auch hier beginnt der Client die Sitzung mit dem Client Hello und der FIDO2 Client Hello Extension. Hier MUSS der Client diesmal den Modus auf FN setzen [1].
Der Server überträgt nach dem Server Hello die FIDO2 Name Request-Nachricht, welche 32 Zufallsbytes S enthält. Nachdem der Client die Certificate-, Certificate Verifiy-, und Finished-Nachrichten des Servers erhalten und überprüft hat, sendet auch er 32 Zufallsbytes C und den benötigten Nutzernamen als FIDO2 Name Response an den Server. Da beide Nachrichten nach dem Server Hello gesendet werden, sind beide bereits durch Verschlüsselung geschützt, weshalb ein Dritter den Nutzernamen nicht mitlesen kann [1].
Beide können, nachdem sie den entsprechenden Zufallsanteil erhalten haben, einen ephemeren Nutzernamen als SHA256(S||C) berechnen. Diese Vorgehensweise bewirkt, dass keiner der beiden Endpunkte den ephemeren Nutzernamen alleine berechnen kann. Angreifern wird es so erschwert, durch Überschreibung des ephemeren Nutzernamens eine Wirkung zu erzielen [1].
Der Server speichert nun ein Tripel T = (ephemerer Nutzername, Nutzername, Lebensdauer). Der ephemere Nutzername ist damit dem Nutzernamen zugeordnet. Die Lebensdauer DARF NICHT mehr als sieben Tage betragen, SOLLTE aber nur wenige Minuten betragen [1]. Allerdings ergibt eine Lebensdauer von sieben Tagen für eine von Breitkopf entwickelte und auch hier später beschriebene und verwendete Modifikation Sinn. Falls der Nutzer nicht unter dem angegebenen Nutzernamen registriert ist, MUSS der Handshake trotzdem normal

weiterlaufen [1]. Denn sonst wäre es für einen Angreifer möglich, den Registrierungsstatus eines Nutzers zu erfragen.
Nach dem 1. Handshake MUSS über eine neue TCP-Verbindung der zweite Handshake ausgeführt werden [1].

Im zweiten Handshake, der in großen Teilen analog zum Handshake im FI-Modus ist, sendet der Client erneut das Client Hello mit der FIDO2 Client Hello Extension. Auch hier MUSS der Modus auf FN gesetzt sein. Allerdings fügt der Client der Client Hello Extension diesmal den zuvor ausgehandelten ephemeren Nutzernamen hinzu [1].
Der Server ermittelt nun den zugehörigen Nutzernamen. Er MUSS die Gültigkeit des ephemeren Nutzernamens dafür überprüfen. Ist er abgelaufen oder in der Datenbank des Servers nicht vorhanden, KANN der Server auf das Senden einer FIDO2 Name Request-Nachricht zurückfallen. Alternativ KANN er den Handshake mit einem „fido2_bad_request“-Alert beenden. Anschließend MUSS der ephemere Nutzername nach einmaliger Verwendung gelöscht werden [1].
Nach dem Server Hello sind alle weiteren Nachrichten „forward secure“ verschlüsselt. Der Server sendet nun, wie im FI-Modus, die FIDO2 Assertion Request-Nachricht, die neben allen weiteren Parametern eine Liste mit den notwendigen Allow-Credentials enthält. Ist der Nutzername nicht registriert, MUSS der Server trotzdem eine FIDO2 Assertion Request-Nachricht senden, um den Registrierungsstatus zu schützen [1]. Eine Möglichkeit der Umsetzung wird in Abschnitt 7 gezeigt.
Die FIDO2 Assertion Response-Nachricht muss das „user_handle“-Feld nun nicht beinhalten, KANN sie aber [1].
Nachdem der Server die Antwort des Clients verifiziert hat, können sicher Daten ausgetauscht werden.

6 Auswahl einer Kryptobibliothek

Nachdem die theoretischen Aspekte und die Vor- und Nachteile der TLS-Erweiterung mit FIDO2 aufgezeigt wurden, soll nun eine passende Bibliothek für den neuen praxisnähren PoC ausgewählt werden. Basierend auf der Implementierung von Breitkopf und allgemeinen Umständen spielen dafür verschiedene Kriterien eine Rolle:

1. Die Bibliothek muss TLS 1.3 unterstützen.
2. Die Validierung von Zertifikaten soll möglich sein.
3. Der Code soll kompakt, aber dabei übersichtlich sein.
4. Die Bibliothek soll gut erweiterbar sein.
5. Zusätzliche Hilfsmittel müssen technisch kompatibel sein.
6. Der Code sollte Open Source sein, um diesen den am PoC interessierten Nutzern bereitzustellen zu können, zum Beispiel über ein Git-Repository.

6.1 Programmiersprache

In Bezug auf Anforderung 5 ergeben sich bereits einige Einschränkungen. Da eine Kommunikation per CTAP mit einem Token auf der Client-Seite während des Handshakes unerlässlich ist, ist eine zusätzliche Bibliothek erforderlich, die CTAP unterstützt.
Breitkopf nutzte zu diesem Zweck die python-fido2-Bibliothek von Yubico [1]. Um diese mit einer passenden Kryptobibliothek zu kombinieren, ist allerdings eine passende Bibliothek in Python nötig, die es unter den gängigen Implementationen nicht gibt [25].
Nach Recherchen scheint die einzige plausible Alternative die Nutzung der libfido2-Bibliothek von Yubico zu sein. Sie unterstützt die nötigen Standards und ist in C implementiert [26].
Aus Gründen der Praktikabilität werden daher im Folgenden nur Kryptobibliotheken in C berücksichtigt. Das ist zwar eine Einschränkung, jedoch ist C eine systemnahe Sprache, die das Potential hat, wesentlich effizientere Implementationen zu erzielen als Python, die eine Interpreter-Sprache ist [27].

Die notwendigen Operationen auf der Seite des Servers, wie das Verifizieren von Responses und die Erzeugung von Challenges, schränken die Auswahl nicht zwangsläufig ein, weil im Zweifel auch ein separater WebAuthn-Server genutzt werden kann, der die Authentifizierung im Zusammenspiel mit dem eigentlichen TLS-Server durchführt.

6.2 Auswahl eines Kandidaten

Mit einer von den Kriterien abhängigen Vorauswahl sollen nun drei Bibliotheken vorgestellt und bewertet werden. Anschließend wird eine von ihnen für die TLS-Erweiterung mit FIDO2 ausgewählt. Hier ist zu beachten, dass die Einschätzung der Erweiterbarkeit bezüglich der Übersichtlichkeit des Codes und anderen Parametern eine Ermessensfrage ist. In allen drei folgenden Bibliotheken wäre die Erweiterung mit FIDO2 prinzipiell möglich.

6.2.1 OpenSSL

OpenSSL ist eine der bekanntesten Bibliotheken, die TLS implementiert. Das Projekt startete bereits 1998 [28]. Neben einer TLS-Implementation ist OpenSSL auch für allgemeine kryptografische Zwecke zu gebrauchen [29]. Die Quellen von OpenSSL sind in C mit teilweise eingebettetem Assembly programmiert.
OpenSSL besteht aus vier wesentlichen Hauptkomponenten. Als Grundpfeiler dient die libcrypto, die zahlreiche Kryptoprimitive implementiert [30].
Neben der libcrypto ist die Engine für die Bereitstellung weiterer kryptografischer Funktionalität verantwortlich. Hier können unter anderem externe Quellen mit eingebunden werden [30].
In der libssl ist die eigentliche TLS-Implementation untergebracht. Dazu gehören des Weiteren ältere SSL-Versionen und DTLS [30].
Die letzte Hauptkomponente bilden die Applications. Sie stellen Anwendungen, die sich aus den anderen Komponenten ergeben, zur Verfügung. Dazu gehören unter anderem Tools zur Zertifikatgenerierung, TLS-Test-Tools und Tools zur Schlüsselerzeugung [30].
Aufgrund der vielen unterschiedlichen Anwendungsmöglichkeiten handelt es sich bei OpenSSL um eine recht umfangreiche Bibliothek, die aktuell aus mehr als 650.000 Codezeilen besteht [32]. Davon entfallen mehr als 70.000 Codezeilen auf die TLS-Implementation [33].
Wie viele Protokoll-Implementationen und insbesondere diese, die TLS implementieren, ist die libssl in OpenSSL als sogenannte Zustandsmaschine konzipiert. Abstrakt erklärt heißt das, dass Client und Server von Beginn an einen Zustand annehmen und abhängig von den Nachrichten, die sie erhalten, den Zustand wechseln. OpenSSL differenziert zwischen einer Zustandsmaschine, die für das Erhalten und Versenden von Nachrichten zuständig ist und einer, die für den Handshake insgesamt verantwortlich ist [31].
Wird eine Handshake-Nachricht erhalten, wird sie entsprechend ihres Typs von der Zustandsmaschine für den Nachrichtenfluss gelesen und die für diese Nachricht spezifische Callback-Funktion ausgeführt. Je nach Ergebnis ändert sich dann der Zustand in der Handshake-Zustandsmaschine, die wiederum ggf. das Senden einer Handshake-Nachricht auslöst [31].
Um weitere Handshake-Nachrichten hinzuzufügen, müssten der Zustandsmaschine deshalb neue Zustände hinzugefügt und passende Callback-Funktionen für das Senden und Empfangen ergänzt werden.
In der folgenden Tabelle wird die Bibliothek nach den einzelnen Kriterien bewertet:

Kriterium	Erfüllung/ Anmerkung
TLS 1.3	OpenSSL unterstützt TLS in der Version 1.3 [31].
Validierung von Zertifikaten	OpenSSL unterstützt die Validierung von Zertifikaten [31].
Code	Mit mehr als 70.000 Codezeilen ist der TLS-Code nicht sehr kompakt. Er ist jedoch funktional in mehrere Quelldateien strukturiert [31].
Erweiterbarkeit	OpenSSL ist ein Community-Projekt und daher auch für eine Erweiterbarkeit offen. Der Code ist als Zustandsmaschine konzipiert und so auf Erweiterbarkeit ausgelegt [31].
Hilfsmittel	OpenSSL ist mit anderen Hilfsmitteln, die für die Erweiterung notwendig sind, kombinierbar.
Open Source	OpenSSL ist unter einer Apache-Lizenz 2.0 Open Source [25].

Tabelle 6.2.1: Bewertung von OpenSSL bzgl. der Kriterien

Als Zwischenfazit lässt sich festhalten, dass OpenSSL durchaus für die Erweiterung mit FIDO2 geeignet ist. Die Konzeption als Zustandsmaschine erleichtert es, weitere Nachrichtentypen und Client Hello Extensions hinzuzufügen. Trotzdem ist die Struktur des Codes nicht trivial. OpenSSL ist eine sehr große Bibliothek, die viele Funktionalitäten anbietet. Für eine Bachelorarbeit muss auch immer der zeitliche Rahmen beachtet werden, weshalb es sich durchaus lohnen kann, auch andere kleinere Bibliotheken in Betracht zu ziehen.

6.2.2 WolfSSL

WolfSSL ist ebenfalls eine TLS-Bibliothek, die aus einem Projekt hervorging, das 2004 seinen Anfang nahm [34].
Die Idee war es, eine Alternative zum bereits erschaffenen OpenSSL zu kreieren. Dabei wollte man verschiedenen Ansprüchen gerecht werden. Zum einen sollte die Bibliothek portabler, kleiner und schneller sein und zum anderen eine moderne API zur Verfügung stellen [34]. So kam es, dass wolfSSL gerade für eingebettete Systeme, Mikrocontroller und andere kompakte Umgebungen Verwendung findet. Die Entwickler von wolfSSL werben damit, dass wolfSSL 20-mal kleiner ist als OpenSSL und auch mit geringen Hauptspeicherkapazitäten arbeiten kann [34].
Auch wolfSSL besteht im Kern aus mehreren Komponenten. Als kryptografische Stütze fungiert die wolfCrypt-Bibliothek, die diverse Kryptoprimitive implementiert. WolfSSL an sich implementiert die TLS-Funktionalitäten. Als letzte Komponente gibt es einen „wolfSSL OpenSSL Compability Layer“. Dieser soll die Kompatibilität mit OpenSSL erleichtern [35].

Die Architektur des Codes für einen TLS-Handshake folgt einem ähnlichen Prinzip wie OpenSSL.
Auch hier findet der Handshake über eine Zustandsmaschine statt, die als Switch-Statement mit Fallthrough konzipiert ist. Wird eine Handshake-Nachricht erhalten, wird sie entsprechend verarbeitet und anschließend ggf. der Zustand gewechselt [36].

Einziger großer Unterschied ist die Kompaktheit des Codes. Fast der gesamte TLS 1.3-Handshake ist hier mit über 8500 Zeilen in einer C-Quelldatei implementiert [36]. Das hat auf der einen Seite den Vorteil, dass die Bibliothek wesentlich kleiner ist, sprich das Nutzungsargument für wolfSSL. Auf der anderen Seite ist der Code so recht schwierig zu analysieren und macht eine Erweiterung aufwendiger.
In der folgenden Tabelle wird die Bibliothek nach den einzelnen Kriterien bewertet:

Kriterium	**Erfüllung/ Anmerkung**
TLS 1.3	WolfSSL unterstützt TLS in der Version 1.3 [35].
Validierung von Zertifikaten	WolfSSL unterstützt die Validierung von Zertifikaten [36].
Code	Der Code ist bis laut Webseite von wolfSSL bis zu 20-mal kleiner als OpenSSL, sprich sehr kompakt [37].
Erweiterbarkeit	Auch wolfSSL ist ein Community-nahes Projekt, welches zur Erweiterung und Mitarbeit ermutigt. Allerdings gestaltet sich die Erweiterung als eher kompliziert, da der Code aufgrund der Kompaktheit weniger gut strukturiert ist [36].
Hilfsmittel	WolfSSL ist mit anderen Hilfsmitteln, die für die Erweiterung notwendig sind, kombinierbar.
Open Source	WolfSSL ist unter einer GPLv2-Lizenz Open Source [25].

Tabelle 6.2.2: Bewertung von wolfSSL bzgl. der Kriterien

WolfSSL ist eine kompakte und schnelle Alternative zu OpenSSL, deren Ziel es ist, eine portable Implementation von TLS zur Verfügung zu stellen. Obwohl technisch gesehen alle Voraussetzungen für eine Erweiterung mit FIDO2 gegeben sind, ist der Code weniger gut strukturiert, weshalb hier mit einer längeren Einarbeitungszeit zu rechnen wäre.
Es gilt nun eine Bibliothek zu finden, die zwar einen möglichst kompakten Code anbietet, jedoch besser strukturiert ist.

6.2.3 GnuTLS

GnuTLS ist eine Bibliothek in C zur sicheren Datenübertragung mit TLS bzw. DTLS [38]. Die erste dokumentierte Version im Newslog des dazugehörigen Git-Repositories erschien bereits im Jahr 2000 [40]. GnuTLS ist im Funktionsumfang ähnlich zu OpenSSL, hat jedoch eine andere Lizenz [38] und ist deshalb mit mehr Software-Projekten kompatibel.
GnuTLS besteht aus drei wesentlichen Hauptkomponenten. Im „TLS protocol part“ ist das eigentliche Protokoll vollständig implementiert. Der „Certificate part“ ist für das Verarbeiten und Validieren von Zertifikaten zuständig und nutzt dafür den Funktionsumfang der libtasn1-Bibliothek. Zuletzt hat auch GnuTLS ein „Cryptographic back-end“ bestehend aus den Bibliotheken Nettle und gmplib, das diverse Kryptoprimitive zur Verfügung stellt [39].

GnuTLS nutzt ebenfalls eine Zustandsmaschine für die Implementierung des Handshakes. Es handelt sich um das gleiche Prinzip wie bei OpenSSL und wolfSSL. Im Fall von GnuTLS gibt es

eine Zustandsmaschine für die Versionen, die älter sind als TLS 1.3. Wenn bei dieser festgestellt wird, dass die Version 1.3 genutzt werden soll, wird in eine andere Zustandsmaschine gewechselt, wo anschließend der Handshake für TLS 1.3 ausgeführt wird [41]. Die Funktionalitäten sowie zusätzliche Handshakenachrichten sind im Ordner „tls13“ untergebracht. Für neue Nachrichten kann hier einfach eine neue C-Quelldatei eingefügt werden, die die entsprechenden Funktionen zum Senden und Erhalten implementiert.
Die Dokumentation von GnuTLS ist sehr umfangreich und klar gegliedert. Es gibt sogar eine Sektion, in der beschrieben wird, wie man neue Client Hello Extensions hinzufügen kann, was ebenfalls signifikant für die Erweiterung mit FIDO2 ist [39].
In der folgenden Tabelle wird die Bibliothek nach den einzelnen Kriterien bewertet:

Kriterium	**Erfüllung/ Anmerkung**
TLS 1.3	GnuTLS unterstützt TLS in der Version 1.3 [38].
Validierung von Zertifikaten	GnuTLS unterstützt die Validierung von Zertifikaten [38].
Code	Der Code ist übersichtlich und funktional strukturiert [41].
Erweiterbarkeit	Eine separate Zustandsmaschine für TLS 1.3 ermöglicht eine gezielte Erweiterung. Es liegt ebenfalls eine gute Dokumentation vor, die Hinweise für eine Erweiterung enthält [41].
Hilfsmittel	GnuTLS ist mit anderen Hilfsmitteln, die für die Erweiterung notwendig sind, kombinierbar.
Open Source	GnuTLS ist unter einer LGPLv2.1+-Lizenz Open Source [25].

Tabelle 6.2.3: Bewertung von GnuTLS bzgl. der Kriterien

Von den vorgestellten Bibliotheken ist GnuTLS nach eingehender Prüfung für diese Erweiterung die geeignetste und wird damit ausgewählt. Der Code ist klar strukturiert und bereits nach einer relativ kurzen Einarbeitungsphase ist der Kontrollfluss der Bibliothek klar erkennbar. Es lässt sich zügig ausmachen, an welchen Stellen Modifikationen erfolgen müssen, um eine Erweiterung einzufügen.
Dass die Lizenz theoretische eine größere Verbreitung möglich macht, ist ebenfalls von Vorteil.

Sicherheitslücke

Im Juni 2020 wurde bei GnuTLS eine große Sicherheitslücke aufgedeckt. Sie betrifft die Verwendung von Session-Tickets zur Fortsetzung einer Sitzung. Es wurde eine zusätzliche Schlüsselrotation in das Schlüsselmaterial von Session-Tickets eingefügt, die Null-Schlüssel verursachte.
Bei TLS 1.3 konnte ein Man-in-the-Middle so Datenverkehr live entschlüsseln. Bei TLS 1.2 ließen sich aufgezeichnete Sitzungen im Nachhinein entschlüsseln [42].
Dieser Fehler ist in der Version 3.6.14 behoben. Die Erweiterung mit FIDO2 wird deshalb in keiner älteren Version implementiert.

7 PoC-Implementation

Im vorigen Abschnitt wurde die Bibliothek GnuTLS für die Erweiterung mit FIDO2 ausgewählt. Dieser Abschnitt beschreibt die neue PoC-Implementation und wertet sie aus.

7.1 Anforderungen

Der Erfolg der Implementierung soll an bestimmten Anforderungen gemessen werden, die teilweise auch auf den von Breitkopf formulierten Anforderungen basieren.

Die wichtigste Anforderung ist, dass die Sicherheit der Protokolle TLS und FIDO2 jeweils nicht gefährdet sein darf. Der TLS-Handshake muss weiterhin korrekt ablaufen und darf keine unerwarteten Zustände annehmen. Auch die Authentifizierung mit FIDO2 darf nicht beeinträchtigt werden.
Des Weiteren soll die Latenz durch hinzugefügte Funktionalität nicht zu hoch werden. Ein Handshake mit FIDO2-Authentifizierung soll trotzdem weiterhin keinen Timeout verursachen. Dass die neuen Nachrichten die Größe betreffend innerhalb der Grenzen liegen, hat Breitkopf bereits gezeigt. Auch der Handshake an sich dauert bei ihm nicht zu lang [1]. Hier wird dies noch einmal evaluiert.
Die Erweiterung mit FIDO2 soll die Usability der API von GnuTLS nicht verschlechtern. Es soll also eine nutzerfreundliche Verwendung möglich sein.
Zuletzt soll die Spezifikation von Breitkopf inkl. des Formats der Nachrichten möglichst eins zu eins übernommen werden, sodass der Server aus Breitkopfs Implementation zum Schluss mit dem Client der neuen Implementation kompatibel ist und umgekehrt. Das trägt zum erfolgreichen Test des neuen PoCs bei.

7.2 Aufbau

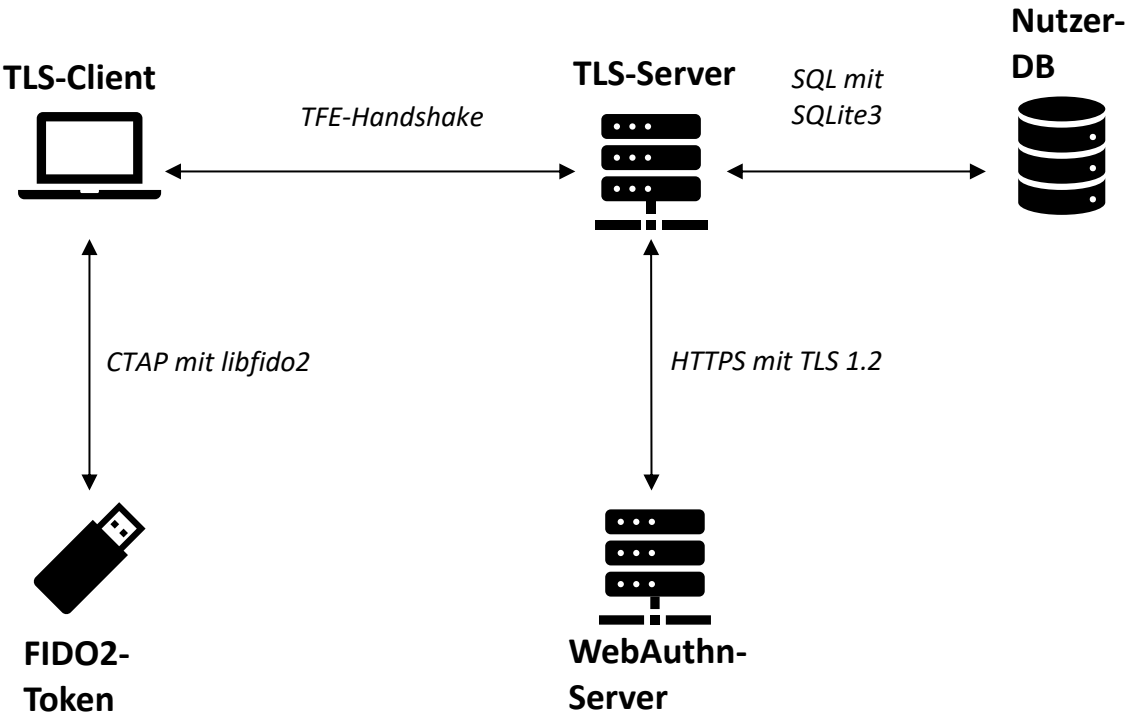

Abbildung 7.2: Aufbau der PoC-Implementation
Quelle: Eigene Abbildung

Wie der Abbildung 7.2 entnommen werden kann, werden die TLS-Funktionalität und die WebAuthn-Server-Funktionalität auf jeweils einem anderen Server ausgeführt. Das ist konform zur Spezifikation von Breitkopf, der offen lässt, ob der TLS-Server auch die WebAuthn-Funktionalität direkt übernimmt oder das in Kommunikation mit einer weiteren Anwendung tut [1]. Zum Zeitpunkt der Arbeit war es leider nicht möglich, eine C-Bibliothek zu finden, die Funktionen für einen WebAuthn-Server zur Verfügung stellt. Der Verbund aus TLS-Server und WebAuthn-Server, die in den meisten Fällen auf einem System laufen werden, ist als RP zu betrachten.
Stattdessen kommt hier der java-webauthn-server von Yubico zum Einsatz. Über eine Webschnittstelle können bei ihm neue Nutzer registriert werden, sowohl mit resident Credentials als auch mit nicht nicht resident Credentials. Der eigentliche TLS-Server und der WebAuthn-Server kommunizieren über HTTPS mit TLS 1.2. Der java-webauthn-server unterstützt TLS 1.3 nicht, was allerdings keine Sicherheitseinbußen zur Folge hat, da TLS 1.2 nach wie vor weit verbreitet ist und bei richtiger Konfiguration als sicher gilt [43]. An dieser Stelle wird HTTPS standardmäßig verwendet, weil der java-webauthn-server so implementiert ist. In der Praxis kommt es lediglich darauf an, dass es einen sicheren Kanal zwischen TLS-Server und Webauthn-Server gibt. Das kann auch durch eine exklusive Verkabelung, VPN oder z.B. indem beide Prozesse auf einem Rechner laufen, erreicht werden. In ersterem und letzterem Fall werden so TLS-Handshakes überflüssig und es kann viel Latenz eingespart werden.

Der TLS-Server verwaltet die Tripel T = (ephemerer Nutzername, Nutzername, Lebensdauer) in einer eigenen SQLite3-Datenbank. Auch die Kommunikation mit dieser erfolgt über SQLite3. Auch andere Lösungen zur Erstellung von Datenbanken und der Durchführung von Operationen auf dieser wäre an dieser Stelle möglich gewesen.

Für die Kommunikation mit dem FIDO2-Token wird die schon erwähnte libfido2 von Yubico verwendet. Sie unterstützt CTAP1 und CTAP2.

Als zusätzliches Hilfsmittel dient die C-Jansson-Bibliothek, die diverse Operationen auf JSON-Objekte anbietet. Das ist zum einen nötig, weil der WebAuthn-Server JSON-Objekte sendet, die verarbeitet werden müssen und auch selbst JSON-Objekte vom TLS-Server erwartet. Zum anderen muss der Client auch JSON-Objekte erzeugen und serialisieren, um die „clientData“ zu erzeugen.
Außerdem wird auf die ohnehin schon GnuTLS verwendete Bibliothek Nettle zurückgegriffen, um Base64url-Codierungen durchzuführen.
Für URL-Codierungen wird die Bibliothek libcurl verwendet.

7.3 Aspekte der Implementierung

Für der Implementierung wurde die stabile Version 3.6.15 von GnuTLS verwendet. Das zugehörige Git-Repository ist unter [44] zu finden. Die Datei README.md enthält eine kurze Beschreibung und Hinweise zur Installation.

7.3.1 Die Registrierung

Wie bereits erläutert, ist die Einbindung der Registrierung in den TLS-Handshake nicht Teil dieser Arbeit und wird für den TFE-Handshake vorausgesetzt. Trotzdem muss für eine Ausführung des Handshakes vorab eine Registrierung stattgefunden haben.
Dazu ist der java-webauthn-server von Yubico zu starten, was vor der Ausführung des Handshakes sowieso notwendig ist. Anschließend kann unter https://localhost:<PORT> eine Registrierung mit resident oder mit nicht resident Credentials vorgenommen werden. Standardmäßig ist der Port 8443. Er kann aber auch über eine Umgebungsvariable geändert werden.
Die Nutzer sind dann entsprechend auf dem Server hinterlegt. Der Handshake kann nun erfolgreich mit dem zur Registrierung verwendeten Token durchgeführt werden.

7.3.2 Der Handshake

Die neue PoC-Implementierung unterstützt den TFE-Handshake im FI-Modus und im FN-Modus, sowohl als doppelten als auch als verkürzten Handshake.

Nutzer-DB

Der Server speichert, etwas anders als ursprünglich definiert, ein Tripel T = (ephemerer Nutzername, Nutzername, Ablaufdatum), da es einfacher ist, vor dem Datenbankzugriff im ersten Handshake das Ablaufdatum zu berechnen, als im zweiten Handshake mit Hilfe der Lebensdauer das Ablaufdatum zu berechnen. Liefert der Client in der Client Hello Extension einen ephemeren Nutzernamen, kann sofort überprüft werden, ob das Ablaufdatum vor dem aktuellen Datum liegt und dann entsprechend reagiert werden. Ist ein ephemerer Nutzername abgelaufen oder nicht vorhanden, wird der Handshake mit einem „fido2_bad_request“-Alert beendet, statt auf das Senden eines FIDO2 Name Requests zurückzufallen. Die Lebensdauer eines ephemeren Nutzernamens beträgt aufgrund der Unterstützung des verkürzten Handshakes sieben Tage.
Unter /doc/examples ist das C-Programm „fido2-user-db.c“ hinterlegt, das bei Ausführung eine passende SQLite3-Datenbank generiert.

Schutz des Registrierungsstatus

Bevor der Server eine FIDO2 Assertion Request-Nachricht schickt, führt er einen TLS-Handshake mit dem WebAuthn-Server durch, um anschließend über einen HTTPS-Post auf dem Pfad „api/v1/authenticate“ die „PublicKeyCredentialRequestOptions“ als JSON-Objekt zu erhalten. Im FI-Modus muss kein Nutzername übergeben werden, da die „PublicKeyCredentialRequestOptions“ in diesem Fall nutzerunabhängig sind. Im FN-Modus wird der Nutzername x-www-form-urlencoded übergeben und die „PublicKeyCredentialRequestOptions“ enthalten eine Liste mit Allow-Credentials.
Ist kein Nutzer unter dem angegebenen Namen registriert, antwortet der WebAuthn-Server entsprechend. Wie in Abschnitt 5 beschrieben, muss in einem solchen Fall der Registrierungsstatus geschützt werden. Der Server schickt dafür also trotzdem eine FIDO2 Assertion Request Nachricht mit einer quasi-zufälligen base64url-codierten Challenge, die dieselbe Länge hat wie diese, die vom WebAuthn-Server erzeugt werden. Der Wert „user_verification“ ist auf „preferred“ gesetzt, was auch der WebAuthn-Server standardmäßig

vornimmt. Die Erzeugung von quasi-zufälligen Allow-Credentials gestaltet sich als etwas komplizierter. Das hat den Grund, dass bei einem gleichen Nutzernamen auch wieder und wieder die gleichen Allow-Credentials erzeugt werden müssen. Der Nutzername muss also als Seed für einen Quasi-Zufallsgenerator verwendet werden. Doch auch das reicht nicht, denn ein Angreifer könnte selbst den Nutzernamen als Seed für diesen Zufallsgenerator verwenden und das Ergebnis mit der Allow-Credential-ID abgleichen. Daher muss der Server in die Zufallsberechnung ein nur ihm bekanntes Geheimnis einfließen lassen, welches ihm als Argument bei der FIDO2-Konfiguration übergeben wird. Die ID der Allow-Credentials wird dann mit Hilfe der SHA3-Shake256-Funktion aus Nettle generiert. Diese funktioniert wie eine kryptografische Hashfunktion, mit dem Unterschied, dass Ausgaben variabler Länge erzeugt werden können. Sie erhält als Eingabe die Konkatenation aus Nutzername und Geheimnis des Servers. Heraus kommt eine Allow-Credential-ID mit der gleichen Länge wie sie der WebAuthn-Server erzeugt. Eine auf diese Weise erzeugte FIDO2 Assertion Request-Nachricht ist von der normalen kaum zu unterscheiden. Diese Maßnahme macht es zumindest sehr schwierig für einen Angreifer, den Registrierungsstatus eines Nutzernamens zu erfahren. Es ist allerdings an dieser Stelle nicht ausgeschlossen, dass er über einen Seitenkanalangriff doch zu seinem Ziel kommt und z.B. anhand unterschiedlicher Laufzeiten den Registrierungsstatus feststellen kann. Es wäre also in weiteren Versionen denkbar, eine Quasirandomisierung der Laufzeit einzubauen, um auch einen Seitenkanalangriff deutlich zu erschweren.

Extensions

Der Einfachheit halber werden im TFE-Handshake weder Client- noch Authentikator-Extensions berücksichtigt. Eine Beeinträchtigung der Sicherheit besteht dadurch nicht. Um kompatibel zu bleiben, werden Extensions entsprechend aus den Nachrichten entnommen, jedoch nicht weiterverarbeitet.

Kommunikation per CTAP

Beim Erhalt der FIDO2 Assertion Request-Nachricht verarbeitet der Client die Parameter entsprechend bevor er mit Hilfe der libfido2 die `authenticatorGetAssertion()`-Methode aufruft. Hier kam es zu Komplikationen, da die libfido2 implizit auf OpenSSL basiert. Ein Header konnte in die entsprechende C-Quelldatei nicht eingebunden werden, weil es sonst zu Typkonflikten zwischen OpenSSL und GnuTLS kommen würde. Daher wurde eine separate C-Quelldatei mit Wrapperfunktionen geschrieben. Das Problem konnte auf diese Weise überbrückt werden.
Wenn der Nutzer die `authenticatorGetAssertion()`-Methode abbricht, dies gilt insbesondere, wenn es für den Test auf Nutzeranwesenheit oder Nutzerverifizierung einen Timeout gibt, wird der Handshake mit einem „user_canceled“-Alert abgebrochen. Falls die `authenticatorGetAssertion()`-Methode aus einem anderen Grund fehlschlägt, wird mit einem „internal_error“-Alert abgebrochen.

Validierung der FIDO2 Assertion Response-Nachricht

Erhält der Server die FIDO2 Assertion Response-Nachricht, muss er das Ergebnis als JSON-Objekt verpacken und per HTTPS an den WebAuthn-Server weiterleiten. Dafür wird eine Session Resumption der vorherigen Sitzung bezüglich des Erhalts der „PublicKeyCredentialRequestOptions“ durchgeführt, um Overhead zu sparen. Die TLS-

Verbindung kann in der Zwischenzeit aufgrund der Implementierung des WebAuthn-Servers nicht offengehalten werden. Es wäre ebenfalls denkbar gewesen, eine Session für die gesamte Laufzeit des Servers zu verwalten und auch bei Beginn des Handshakes immer wieder eine Session Resumption durchzuführen. Allerdings hätten diese Session und die dazugehörigen Daten dann außerhalb des eigentlichen Handshakes verwaltet werden müssen. Aus Sicht des Autors war die andere Variante eine nutzerfreundlichere Lösung.
Der Server muss nun die Antwort des WebAuthn-Servers abwarten. Ist der Status-Code der Antwort 200 (OK), ist die Authentifizierung erfolgreich gewesen, wenn nicht, ist sie fehlgeschlagen. In diesem Fall wird der Handshake mit einem „fido2_authentication_error“ beendet und nicht, wie alternativ vorgeschlagen, der Handshake fortgeführt. Für eine PoC-Implementierung ist das ausreichend.

PSKs

Die Verwendung eines PSKs zur Authentifizierung ist im ersten FN-Handshake möglich, ist jedoch im zweiten FN-Handshake und im FI-Handshake verboten, da die Sicherheit einer authentifizierten Verbindung nicht von einer eventuell unauthentifizierten Verbindung abhängig sein darf [1].

Kommunikation zur Anwendungsschicht

Breitkopf machte in seiner PoC-Implementierung der Anwendungsschicht ein Zertifikat nach erfolgreicher FIDO2-Authentifizierung zugänglich, um kompatibel zur Zertifikats-Authentifizierung zu bleiben. Dazu hinterlegte er bei Registrierung gleich ein entsprechendes Zertifikat, welches dann nur abgerufen werden musste [1]. Anwendungen müssen in diesem Fall nur begrenzt an die neue Erweiterung mit FIDO2 angepasst werden. Informationen über den authentifizierten Nutzer wie z.B. Nutzer-ID oder in welchem Modus (FI oder FN) der Nutzer authentifiziert wurde, können dann dem Zertifikat entnommen werden [1]. Sollte die Erweiterung mit FIDO2 in Zukunft ein gängiger Standard sein, kann auf diesen Mechanismus vermutlich verzichtet werden. Diese Art der Zertifikatsausstellung gestaltet sich in der neuen Implementierung als etwas schwieriger, da dazu der java-webauthn-server erheblich verändert werden müsste, was den zeitlichen Rahmen dieser Arbeit überschreiten würde. Man könnte alternativ für jede erfolgreiche Authentifizierung über eine vorgefertigte ASN.1-Struktur ein Zertifikat zur Laufzeit generieren und beispielsweise zeitsparend mit dem ECDSA signieren. Auch die Anpassung an die bisherige Schnittstelle von GnuTLS ist möglich, sodass Anwendungen kaum verändert werden müssten.
Im PoC dieser Arbeit wurde der Einfachheit halber lediglich ein Mechanismus hinterlegt, der der Anwendung bei erfolgreicher Authentifizierung eine Struktur mit der Nutzer-ID, der Request-ID der Authentifizierungsanfrage beim WebAuthn-Server und dem Modus hinterlegt. Es besteht keine Sicherheitseinschränkung.

7.3.3 Einschränkungen

Der vorherige Abschnitt führt bereits zu Einschränkungen in der Implementierung, die bei einem PoC zwangsläufig auftreten.

Thread-Safety

Im hinzugefügten Wrapper werden globale Variablen verwendet. Die Erweiterung mit FIDO2 ist also nicht als thread-safe zu erachten. Dies betrifft jedoch nur den Client, was in der Praxis kaum zu Einschränkungen führt. Es soll trotzdem an dieser Stelle erwähnt sein. Es stellt sich weiterhin ohnehin die Frage, inwiefern mehrere Threads oder auch Prozesse gleichzeitig auf den Token zugreifen können.

Nutzer-DB

Die Nutzer-DB des TLS-Servers ist unverschlüsselt und liegt unter Umständen im selben Ordner wie der auszuführende Server. Sollte es einem Dritten möglich sein, sich Zugriff zum System zu verschaffen, könnte er alle Paare aus ephemeren Nutzernamen und Nutzernamen einsehen und überschreiben. Der Fokus dieser Arbeit liegt jedoch auf Angriffen durch einen Man-in-the-Middle. Daher fällt dieser Sachverhalt nicht ins Gewicht. Es ist außerdem auf einen Überlauf mit Datenbankeinträgen in der Nutzer-DB des TLS-Servers zu achten. Angreifer könnten das nutzen, um eine DoS-Attacke durchzuführen. Details können einer Analyse von Breitkopf entnommen werden [1]. Alternativ kann die Datenbank in-memory angelegt werden, was dazu führen würde, dass gespeicherte ephemere Nutzernamen flüchtig wären.

Überprüfung der RP-ID

Wie bei Breitkopf prüft der Client die RP-ID lediglich auf Gleichheit mit der Domain der RP, da der Test, ob die RP-ID einer registrierbaren Subdomain entsprecht, nicht trivial ist. Eine weitere Möglichkeit wäre es, zu verlangen, dass die RP-ID als DNS-Alias Teil des TLS-Serverzertifikats sein muss.

Zertifikat des WebAuthn-Servers

Der java-webauthn-server von Yubico sendet nur ein Dummy-Serverzertifikat, weshalb eine Authentifizierung theoretisch fehlschlagen würde. Da allerdings in der Praxis TLS- und WebAuthn-Server auf dem gleichen System bzw. hinter der gleichen Firewall laufen, haben sie ein besonderes Vertrauensverhältnis. Wie unter 7.2 bereits angesprochen, wäre eine TLS-Verbindung in diesem Fall nicht nötig. Jedoch setzt der java-webauthn-server eine TLS-Verbindung voraus, was für eine normale FIDO2-Registrierung und -Authentifizierung auch vorgeschrieben ist. Hier besteht in zukünftigen Versionen Verbesserungspotential, um zusätzliche TLS-Handshakes und damit Latenz zu vermeiden.

In-memory-Problematik

Der java-webauthn-server speichert die Nutzerdaten und alle wichtigen Parameter, die bei der FIDO2-Registrierung hinterlegt werden, in einer Datenbank, die in-memory ist. Wird der Prozess des WebAuthn-Servers beendet, ist die Registrierung verloren.

Den TLS-Server betrifft diese Problematik nicht, denn bei ihm sind alle Datenbankeinträge, also paare aus ephemerem Nutzernamen und Nutzernamen, auf der Festplatte gespeichert. Trotzdem würde der TFE-Handshake ohne neue Registrierung des Nutzers fehlschlagen. Zu Zwecken eines PoCs und um die Umsetzbarkeit zu demonstrieren, ist diese DB-Lösung ausreichend. Unter realen Bedingungen kann eine persistente Datenbank genutzt werden.

FIDO2-Geräte

In der PoC-Implementierung werden Werte der FIDO2 Assertion Request-Nachricht im Feld „transports“ nicht weiterverarbeitet. Es wird von einem USB-Token ausgegangen.

7.3.4 Serverseitige Nutzung

Die wichtigste Funktion für die Schnittstelle des Servers ist die `gnutls_fido2_set_server()`-Methode. Sie erhält als Argument eine Konfiguration, die beschreibt, ob die Authentifizierung mit FIDO2 explizit verboten, erlaubt oder notwendig ist. Außerdem erhält sie einen Pointer zu einer Session, welche für die TLS-Verbindung zum WebAuthn-Server genutzt wird. Dazu kommen IP-Adresse und Port des WebAuthn-Servers, ein Pfad zur Nutzer-Datenbank, die RP-ID, die der Server verwenden soll und das Geheimnis, das für die Generierung einer quasi-zufälligen Allow-Credential-ID genutzt werden soll.
Der Nutzer der Bibliothek kann, wenn er möchte, die `gnutls_fido2_generate_secret()`-Methode nutzen, um ein Geheimnis zu generieren. Ansonsten ist ihm die Art der Generierung überlassen. Dabei sollte er beachten, dass eine Geheimnisgenerierung zur Laufzeit bedeutet, dass der Server bei Neustart ein anderes Geheimnis erzeugt und somit auch eine andere Allow-Credential-ID, falls der genannte Nutzer nicht registriert ist. Somit würde sich der Registrierungsstatus verraten.
Nach dem Handshake ist mit den Methoden `gnutls_fido2_active()` und `gnutls_fido2_client_authenticated()` abzufragen, ob der FIDO2 genutzt wurde und ob der Client ggf. authentifiziert wurde. Wurde FIDO2 genutzt und ist der Client nicht authentifiziert, ist die Verbindung zu beenden, da in diesem Fall nur der erste Handshake ausgeführt wurde.
Wurde der Client authentifiziert, können Informationen über die Authentifizierung mit der Methode `gnutls_fido2_get_auth_info()` für die Anwendung abgerufen werden. Die Informationen können nach Ausgabe oder Verarbeitung mit `gnutls_fido2_deinit_auth_info()` wieder deinitialisiert werden.
In [44] unter /doc/examples/ex-serv-fido2.c befindet sich ein Codebeispiel für einen Echo-Server mit FIDO2-Authentifizierung.

7.3.5 Clientseitige Nutzung

Für den Client ist die wichtigste Funktion die `gnutls_fido2_set_client()`-Methode. Ihr werden stellvertretend für den zu verwendenden Modus kein Nutzername (FI-Handshake), ein Nutzername (doppelter FN-Handshake) oder ein ephemerer Nutzername (einfacher FN-Handshake) übergeben. Dazu kommt die Domain des Servers.
Wird der doppelte Handshake ausgeführt, muss der Client nach dem ersten Handshake einen neuen Handshake mit dem ausgehandelten ephemeren Nutzernamen initiieren. Er kann mit `gnutls_fido2_get_eph_user_name()` den ephemeren Nutzernamen aus der alten Session abfragen, um ihn dann in der `gnutls_fido2_set_client()`-Methode einzusetzen.

Damit die Usability für den Client möglichst hoch ist, wurde zusätzlich die `gnutls_fido2_perform_handshake()`-Methode für den Client eingeführt. Neben den Argumenten für die `gnutls_fido2_set_client()`-Methode wird ihr ein Socket-Deskriptor übergeben, IP-Adresse und Port des Servers und ein Flag, ob das Zertifikat des Servers validiert werden soll. Die `gnutls_fido2_perform_Handshake()`-Methode führt den Handshake dann automatisch aus. Im Rahmen der Funktion werden Zertifikat-Credentials als Credentials alloziert. Hier ist sehr wichtig zu erwähnen, dass der Client weder ein Zertifikat übergeben bekommt noch ein Zertifikat an den Server schickt, denn die Authentifizierung erfolgt mit FIDO2. Die Maßnahme dient lediglich dazu, die entsprechenden (EC)DHE-Gruppen zu aktivieren und ist notwendig, damit das Zertifikat des Servers empfangen und validiert werden kann. Es gilt außerdem zu beachten, dass diese Methode im Prinzip nur ein Wrapper ist, der die Nutzerfreundlichkeit erhöhen soll. Er kann nach den Wünschen des Programmierers verändert werden bzw. der Handshake manuell durchgeführt werden. Ein Beispiel befindet sich in [44] unter /doc/examples/ex-client-fido2.

Um Kompatibilität mit der Anwendung auch auf der Seite des Clients zu erreichen, wäre es für zukünftige Versionen denkbar, hier ebenfalls ein virtuelles Zertifikat an GnuTLS zu übergeben, analog zur Authentifizierung mit einem Clientzertifikat. Das Zertifikat enthielte dann Informationen zur Ausführung des TFE-Handshakes.

7.4 Auswertung

7.4.1 Test

Die Erweiterung mit FIDO2 in GnuTLS wurde auf verschiedene Arten und Weisen getestet. Dazu wurde als erstes die eigene Testsuite von GnuTLS durchlaufen, die mehr als 500 Testfälle enthält. Ziel war es sicherzustellen, dass die Funktionsweise der Bibliothek nicht durch die Erweiterung mit FIDO2 eingeschränkt ist. Das wurde erreicht, indem FIDO2 standardmäßig deaktiviert ist, sodass es keine Auswirkungen auf den inneren Zustand hat, wenn FIDO2 nicht explizit aktiviert wird.
Anschließend wurden etwa 20 Black-Box-Tests für Client und Server durchgespielt, bei der die Eingabekombinationen teilweise mittels Äquivalenzklassenanalyse bestimmt wurden. Ziel war es hier, die verschiedenen Szenarien des TFE-Handshakes zu simulieren und zu ergründen, ob Client bzw. Sever jeweils richtig reagieren und ggf. mit den richtigen Alerts die Verbindung beenden.

Ziel der Arbeit ist es auch, zu überprüfen inwiefern die Spezifikation von Breitkopf durch die neue PoC-Implementation eingehalten wurde. Dazu wurden TFE-Handshakes zwischen den beiden PoC-Implementationen durchgeführt. Leider führte dies aus mehreren Gründen nicht zu einem zufriedenstellenden Ergebnis. Zum einen wird der zum Test verwendete Yubico Security Key nicht von der Implementation von Breitkopf erkannt. Die Vermutung liegt nahe, dass das damit zusammenhängt, dass die alte Version 0.5.0 der Bibliothek python-fido2 zur Kommunikation via CTAP1/2 in seinem Code Verwendung findet [45]. Auf eine neuere Version konnte nicht gewechselt werden, weil in diesem Fall der Code nicht mehr ausführbar wäre. Zum anderen gab es Probleme, weil einige Flags aus den neuen FIDO2-Handshake-Nachrichten anders implementiert waren als in seiner schriftlichen Arbeit angegeben [1, 45]. So konnte nur ein kleiner Teil überprüft werden.

Ein weiteres Augenmerk der Tests lag auf der Vermeidung von Speicherlecks, wofür das Tool valgrind benutzt wurde. Speicherlecks konnten so auf ein Minimum reduziert werden. Im Fall des Clients wurden einige wenige Speicherblöcke von valgrind auf „still reachable“ eingestuft. Bei genauerer Untersuchung konnte sich das Problem auf einen internen Sachverhalt der libfido2 eingegrenzt werden, der von außen nicht ohne Weiteres behoben werden konnte.
Auch beim Server trat ein ähnlicher Fall auf. Allerdings handelt es sich nicht um eine Einschränkung in der Funktionsweise der FIDO2-Erweiterung und es kann auch nicht zu Speicherfehlern kommen.

7.4.2 Benchmark

Für die Auswertung der Implementierung ist es interessant, wie viel langsamer ein TFE-Handshake im Vergleich zu einem herkömmlichen Handshake in der Praxis ist. Es geht zum einen um die Nutzerakzeptanz und zum anderen darum, dass technische Vorgaben wie Timeouts oder maximale Länge der Handshake-Nachrichten eingehalten werden. Dass die maximale Länge von 2^{24}-1 Bytes nicht überschritten wird, hat Breitkopf bereits gezeigt [1]. Was den zeitlichen Rahmen betrifft, gilt es allerdings in diesem Fall den TFE-Handshake neu zu bewerten, weil hier im Gegensatz zur ersten Implementierung die Operationen eines WebAuthn-Servers auf einem separaten Server stattfinden. Durch die zusätzlichen TLS-Verbindungen entsteht so zusätzliche Latenz.
Für die Messungen wurden Handshakes auf dem lokalen Loopback 127.0.0.1 durchgeführt, d.h. Client und Server liefen auf demselben System, um eine Vergleichbarkeit zu schaffen. Beim TFE-Handshake wurde zusätzlich die Zeit, in der die Nutzerverifizierung stattfindet (also der Nutzer eine PIN eingibt oder einen Fingerabdruck zur Freischaltung nutzt), abgezogen, da diese offensichtlich das Ergebnis verfälscht. Die Messungen wurden auf einem Lenovo IdeaPad 520S-14IKB mit einem Intel Core i5 7200U (2,5 GHz, zwei Kerne) ausgeführt. Für jedes Szenario wurden 10 Messungen unter gleichen Bedingungen, also z.B. parallellaufende Prozesse, durchgeführt.
Bei einem normalen TLS-1.3-Handshake ergab sich, dass mit oder ohne Clientauthentifizierung mit Zertifikat, was kaum messbare Unterschiede ergab, die Dauer sich zwischen 3,97 ms und 4,56 ms belief. Ein TFE-Handshake im FI-Modus hatte eine Dauer zwischen 32,627 ms und 35,726 ms. In diesem Szenario war der TFE-Handshake also bis zu 9-mal langsamer. Zustande kommt dies aufgrund eines kompletten zusätzlichen TLS-1.2-Handshakes mit dem WebAuthn-Server und einer Session Resumption dieser Sitzung. Außerdem werden zwei HTTPS-Posts jeweils mit Antwort ausgeführt.
Noch gravierender ist es, wenn der doppelte TFE-Handshake im FN-Modus ausgeführt wird. Dieser nimmt zwischen 54,211 ms und 77,797 ms in Anspruch. Im schlechtesten Fall ist der TFE-Handshake dann bis zu 20-mal langsamer, weil neben den schon genannten Aspekten ein kompletter zweiter Handshake zwischen Client und Server durchgeführt wird.

Die sehr hoch ausfallenden Messergebnisse müssen an dieser Stelle etwas relativiert werden. Da auch der verkürzte Handshake im FN-Modus implementiert ist, muss der doppelte Handshake im Optimalfall mit jedem Client nur einmal ausgeführt werden, es sei denn, der ausgehandelte nächste ephemere Nutzername läuft in der Zwischenzeit ab.
Außerdem muss bedacht werden, dass der Nutzer während des Handshakes eine Nutzerverifizierung durchführen muss, d.h. von einer erhöhten Latenz bekommt dieser wenig mit. In der GnuTLS ist der standardmäßige Timeout auf 40 s gesetzt [41], welcher mit der

FIDO2-Erweiterung in der Regel nicht erreicht wird, solange der Nutzer die Nutzerverifizierung zeitnah durchführt.

7.4.3 Zusammenfassung

Durch die Tests konnten einige Bugs bereinigt werden. Es lässt sich für die Auswertung insgesamt sagen, dass es gelungen ist, eine mit Einschränkungen funktionierende, GnuTLS nicht beeinträchtigende PoC-Implementation zu schreiben. Durch das Hinzufügen eines Wrappers wurde dazu die Nutzerfreundlichkeit erhöht. Probleme mit Timeouts oder einem zu großen Handshake gab es im Testbetrieb ebenfalls nicht. Das Ergebnis des Benchmarkings lag im Rahmen der Erwartungen und attestierte eine genügende Praxistauglichkeit. Lediglich in der Kombination mit der bereitgestellten Implementierung von Breitkopf konnte kein zufriedenstellendes Ergebnis erzielt werden. Dies hatte allerdings keine Unzulänglichkeiten in der neuen Implementation zur Ursache.

7.5 Lizenz

Der erstellte Code soll unter einer passenden Lizensierung Open Source gestellt werden. Er ist ausdrücklich nicht für kommerzielle Zwecke geeignet. Für die Lizenz ist es entscheidend, die verwendeten Bibliotheken anhand ihrer Lizenz einzubeziehen. GnuTLS selbst verwendet für die Bibliothek an sich eine GNU LGPLv2.1+-Lizenz [38].
Die Bibliothek Nettle, welche Grundlage für GnuTLS und auch für die Erweiterung mit FIDO2 ist, verwendet eine duale Lizensierung aus GNU LGPLv3+ und GNU GPLv2+ [46].
Die libfido2 von Yubico verwendet eine BSD-Lizenz [26] und die übrigen Bibliotheken libcurl und Jansson verwenden eine MIT-Lizenz [47, 48], während SQLite3 in der „Public Domain“ platziert ist, also nicht lizensiert ist [49].
Aus den gegebenen Lizenzen und ihrer Kompatibilität ergibt sich, dass sich sowohl eine GNU LGPLv3+-Lizenz als auch eine GPL-Lizenz eignen würden. Da die PoC-Implementation sowieso nicht für einen sicheren Betrieb eingesetzt werden sollte (siehe Abschnitt 7.3.3) und sich somit jegliche kommerzielle Zwecke erübrigen, kommt hier eine GNU GPLv3-Lizenz zum Einsatz, die es nicht erlaubt, eine unter dieser Lizenz gestellte freie Software in eine proprietäre Software zu überführen [50].
Informationen zur Lizenz und die rechtlichen Erwähnungen der verwendeten Bibliotheken sind unter LICENSE bzw. doc/COPYING.FIDO2 in [44] zu finden.

7.6 Ausblick zur Registrierung

Diese Arbeit hat als zweite, unter anderen Bedingungen als die erste, gezeigt, dass eine Einbindung von FIDO2 in TLS als Authentifizierungsmethode praktisch möglich ist. Im Rahmen eines abschließenden Ausblicks soll jetzt betrachtet werden, inwiefern auch die Registrierung zumindest theoretisch sinnvoll in den Handshake miteingebunden werden kann.
Aus technischer Sicht ist eine Einbindung der Registrierung unter Nutzung der bereits entwickelten Mechanismen durchaus möglich. Die FIDO2 Client Hello Extension könnte genutzt werden, um die gewünschte Art der Registrierung anzugeben, also ob die Credentials resident sein sollen und den gewünschten Nutzernamen anzugeben, unter der der Client registriert werden soll. In den Flags der besagten Erweiterung sind in der Spezifikation von

Breitkopf die Bits 1-3 für eine zukünftige Verwendung reserviert [1]. Zwei von diesen könnten für die Registrierungsart genutzt werden.
Allerdings soll auch in diesem Anwendungsfall der Registrierungsstatus des Nutzers geschützt sein. Daher wäre es hier ebenfalls unerlässlich, einen doppelten Handshake durchzuführen. Der ausgehandelte ephemere Nutzername kann dann analog zur Authentifizierung verwendet werden.
Neben der FIDO2 Client Hello Extension könnten zwei weitere Nachrichtentypen implementiert werden. Die FIDO2 Attestation Request-Nachricht enthielte dann sämtliche Parameter der „PublicKeyCreationOptions“, um eine neue PKCS auf dem Authentifikator zu erzeugen. Nachdem der Client alle Parameter verarbeitet und der Authentifikator die neue PKCS generiert hat, würde der Client entsprechend eine FIDO2 Attestation Response-Nachricht an den Server zurücksenden, die die „clientDataJSON“ und das „attestationObject“ enthielte. Nach der Verifizierung durch den Server könnte der Server den Client dann als authentifiziert ansehen. Bei der nächsten Anmeldung wäre dann eine normale Authentifizierung im FI- oder im FN-Modus (auch verkürzt) möglich.

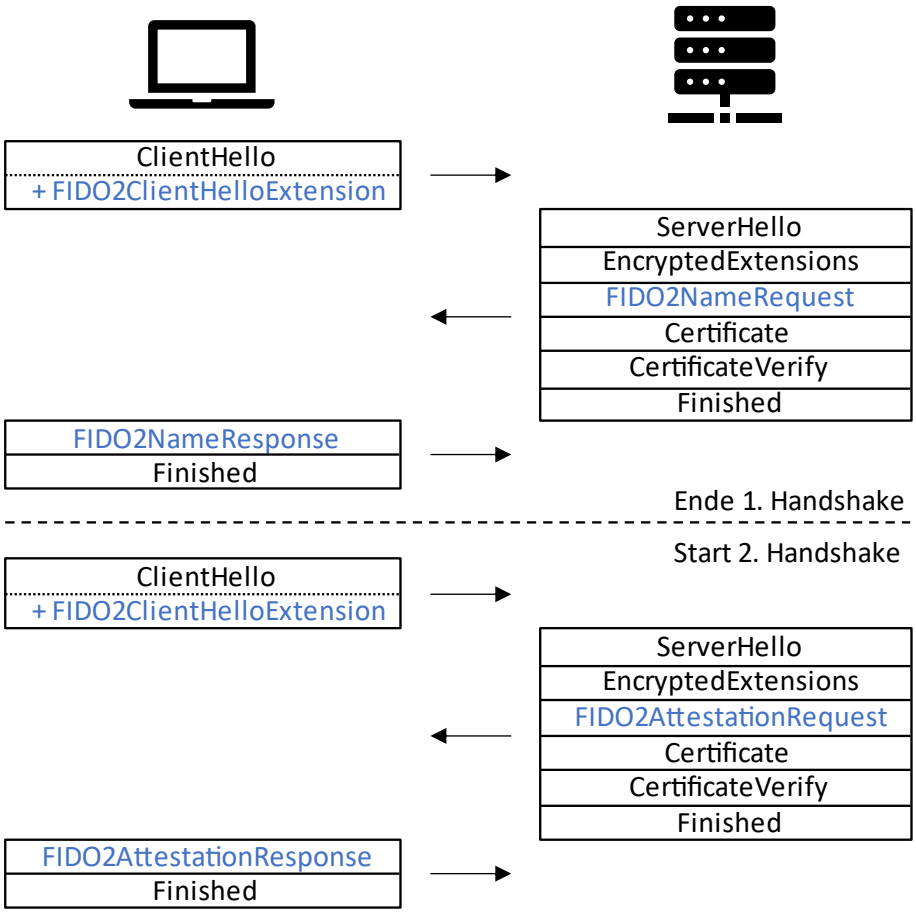

Abbildung 7.6: Registrierung als TFE-Handshake
Quelle: Eigene Abbildung nach [1]

Abbildung 7.6 visualisiert den eben beschriebenen doppelten Handshake noch einmal.

Wie man diesen in der Praxis nutzerfreundlich integriert, beispielsweise im Kontext eines Webbrowsers, ist allerdings eine andere Frage. Wie bereits in Abschnitt 5.2 erläutert, findet die Nutzerauthentifizierung bereits vor dem eigentlichen HTTP-GET statt, weshalb eine Webseite zu diesem Zeitpunkt noch nicht angezeigt ist. Wenn der Nutzer zu diesem Zeitpunkt

noch nicht registriert ist, wäre es denkbar, vorerst einen Handshake durchzuführen, bei dem der Client nicht authentifiziert wird. Statt mit einem Alert abzubrechen, könnte der Server dem Client den Zugang zu ausgewählten Inhalten, wie von Breitkopf bereits vorgeschlagen [1], ermöglichen. Sollte der Nutzer anschließend eine Registrierung wünschen, könnte er über einen passenden Mechanismus eine neue Verbindung und damit den TFE-Handshake zur Registrierung initiieren.
Die Registrierung durch den TFE-Handshake scheint also praktisch umsetzbar zu sein. Ob sich dieses Vorgehen allerdings mehr lohnt als die normale FIDO2-Registrierung, muss im Endeffekt jeder Entwickler für sich entscheiden und hängt vermutlich auch von der Umgebung ab, in der der TFE-Handshake genutzt wird.

8 Fazit

Diese Arbeit hat die ersten Überlegungen von Tom Breitkopf, die Authentifizierung mit FIDO2 in einen TLS-Handshake zu integrieren, fortgeführt. Dazu wurde die Erweiterung mit FIDO2 theoretisch erläutert und dann anschließend eine Kryptobibliothek ausgewählt, um eine neue PoC-Implementation zu schreiben. Ausgewählt wurde GnuTLS, eine entwicklerfreundliche und leicht erweiterbare Bibliothek. Sie ist dazu weit verbreitet und ermöglicht die Validierung von Zertifikaten. Es wurde gezeigt, dass eine Erweiterung mit FIDO2 auch in einer systemnahen Programmiersprache wie C möglich ist, wobei auch Elemente, die eher aus der Webprogrammierung bekannt waren, z.B. JSON-Serialisierungen, in die Implementierung integriert wurden. Für verschiedene Hürden wurden pragmatische Lösungen gefunden.
Es wurden verschiedene Aspekte der Implementierung beleuchtet und auch Einschränkungen benannt. Zu den wichtigen Ergebnissen gehört unter anderem auch die Einführung eines konkreten Mechanismus, der den Registrierungsstatus verschleiert, auch wenn der angegebene Nutzername nicht registriert ist. In der Auswertung konnten solide Ergebnisse erzielt werden, auf denen in weiteren Arbeiten aufgebaut werden kann.
Es konnte außerdem ein konkreter Vorschlag für eine Einbindung der Registrierung in den TFE-Handshake gegeben werden und es wurde weiterhin skizziert, wie die Kompatibilität zu bestehender Anwendungssoftware aufrechterhalten werden kann.

Gerade die letzten beiden Punkte könnte für weiterführende Arbeiten aufgegriffen werden, um das Ziel weiter voranzutreiben, eine nutzerfreundliche und kryptografisch sichere Alternative zu Passwörtern bereitzustellen.

Referenzen

[1] T.-L. J. Breitkopf, *FIDO2 als TLS-1.3-Erweiterung.* Bachelorarbeit, Humboldt-Universität zu Berlin, 2020.

[2] *Trotz „Collection #1-5“: Beim Passwortschutz lernen deutsche Internet-Nutzer nur langsam dazu.* http://web.de/magazine/in-eigener-sache/trotz-collection-1-5-passwortschutz-lernen-deutsche-internet-nutzer-33626818. Zugriff am 02.09.2020.

[3] BSI für Bürger: *Sichere Passwörter erstellen.* https://www.bsi-fuer-buerger.de/BSIFB/DE/Empfehlungen/Passwoerter/passwoerter_node.html. Zugriff am 02.09.2020.

[4] *Passwortmanager sind weit verbreitet.* https://www.com-magazin.de/news/sicherheit/passwortmanager-1167794.html?page=1_passwortmanager-sind-weit-verbreitet. Zugriff am 02.09.2020.

[5] A. Poleshova, *Statistiken zur Internetnutzung in Deutschland.* https://de.statista.com/themen/2033/internetnutzung-in-deutschland/. Zugriff am 02.09.2020.

[6] *Die vier gefährlichsten Schwachstellen bei Authentifizierungsvorgängen.* https://www.itsicherheit-online.com/blog/detail/sCategory/222/blogArticle/3340. Zugriff am 02.09.2020.

[7] FIDO Alliance, *History of FIDO Alliance.* https://fidoalliance.org/overview/history/. Zugriff am 02.09. 2020.

[8] FIDO Alliance, *FIDO Members.* https://fidoalliance.org/members/. Zugriff am 02.09.2020.

[9] LinkFang, *Challenge-Response-Authentifizierung.* https://de.linkfang.org/wiki/Challenge-Response-Authentifizierung. Zugriff am 02.09.2020.

[10] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3.* IETF RFC 8446, 2018.

[11] Wikipedia, *Transport Layer Security.* https://de.wikipedia.org/wiki/Transport_Layer_Security. Zugriff am 03.09.2020.

[12] W. Müller, *IT-Sicherheit Grundlagen.* Vorlesungsfolien. 2020.

[13] M. Slaviero, *TLS/SSL and .NET Framework 4.0.* https://www.red-gate.com/simple-talk/dotnet/net-framework/tlsssl-and-net-framework-4-0/. Zugriff am 03.09.2020.

[14] C. Eckert, *IT-Sicherheit: Konzepte-Verfahren-Protokolle.* München u.a.: De Gruyter Oldenbourg. 9. Auflage. 2014. ISBN: 978-3-486-77848-9.

[15] *OPTLS and TLS 1.3 protocols.* http://cryptowiki.net/index.php?title=OPTLS_and_TLS_1.3_protocols. Zugriff am 04.09.2020.

[16] D. Balfanz Et al, *Web Authentication: An API for accessing Public Key Credentials Level 1.* Technische Spezifikation. 2019.

[17] Yubico, *WebAuthn Client Registration.* https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/WebAuthn_Client_Registration.html. Zugriff am 07.09.2020

[18] FIDO Alliance, *FIDO2: WebAuthn & CTAP.* https://fidoalliance.org/fido2/. Zugriff am 07.09.2020.

[19] FIDO Alliance, *How FIDO works.* https://fidoalliance.org/how-fido-works/. Zugriff am 07.09.2020.

[20] A. Sinitsyna, *Beyond Passwords: FIDO2 and WebAuthn in Practice.* https://www.inovex.de/blog/fido2-webauthn-in-practice/. Zugriff am 07.09.2020.

[21] D. Baghdasaryan Et al, *FIDO Security Reference.* Technische Spezifikation. 2018.

[22] C. Brand Et al, *Client to Authenticator Protocol (CTAP).* Technische Spezifikation. 2019.

[23] Yubico, *WebAuthn Client Authentication.* https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/WebAuthn_Client_Authentication.html. Zugriff am 09.09.2020.

[24] *What is SSL/TLS Client Authentication? How does it work?* https://comodosslstore.com/blog/what-is-ssl-tls-client-authentication-how-does-it-work.html. Zugriff am 18.09.2020.

[25] Wikipedia, *Comparison of TLS implementations.* https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations. Zugriff am 28.09.2020.

[26] Yubico, *libfido2.* https://developers.yubico.com/libfido2/. Zugriff am 28.09.2020.

[27] Python, *General Python FAQ.* https://docs.python.org/3/faq/general.html. Zugriff am 28.09.2020.

[28] OpenSSL, *Newslog.* https://www.openssl.org/news/newslog.html. Zugriff am 28.09.2020.

[29] OpenSSL, *Welcome to OpenSSL!* https://www.openssl.org/. Zugriff am 28.09.2020.

[30] OpenSSL Management Committee, *OpenSSL Strategic Architecture.* https://www.openssl.org/docs/OpenSSLStrategicArchitecture.html. Zugriff am 28.09.2020.

[31] *openssl/openssl.* https://github.com/openssl/openssl. Git Repository. Letzter Commit-Hash: 7d6766cb537e5cebc99e200bc537f744878a87a4.

[32] *OpenSSL.* https://www.openhub.net/p/openssl. Zugriff am 29.09.2020.

[33] *Amazon just wrote a TLS crypto library with only 6,000 lines of C code.* https://www.theregister.com/2015/07/01/amazon_s2n_tls_library/. Zugriff am 29.09.2020.

[34] wolfSSL, *ABOUT US*. https://www.wolfssl.com/about/. Zugriff am 01.10.2020.

[35] wolfSSL, *wolfSSL User Manual.* Technisches Handbuch. Version 4.1.0. 2019.

[36] *wolfSSL/wolfssl.* https://github.com/wolfSSL/wolfssl/. Git-Repository. Letzter Commit-Hash: 20d28e1b65470d599bf33c2d96f692024db9e922

[37] wolfSSL, *wolfSSL Embedded SSL/TLS Library.* https://www.wolfssl.com/products/wolfssl/. Zugriff am 01.10.2020.

[38] GnuTLS, *Welcome to GnuTLS project pages.* https://www.gnutls.org/index.html. Zugriff am 05.10.2020.

[39] N. Mavrogiannopoulos Et al, *GnuTLS manual.* Technisches Handbuch. Version 3.6.15. 2020.

[40] *gnutls/gnutls/blob/master/NEWS.* https://gitlab.com/gnutls/gnutls/blob/master/NEWS. Datei in Git-Repository. Letzter Commit-Hash (auf dieser Datei): f748e8df7f7220656be116f2e354fc3aabbdde67

[41] *gnutls/gnutls.* https://github.com/gnutls/gnutls. Git-Repository. Letzter Commit-Hash: 6f034aa2e9f140626de2b9413715651dffe9e394

[42] J. Schmidt, *Massives Sicherheitsproblem in GnuTLS erlaubt Mitlesen von Kommunikation.* https://www.heise.de/security/meldung/Massives-Sicherheitsproblem-in-GnuTLS-erlaubt-Mitlesen-von-Kommunikation-4779992.html. Zugriff am 05.10.2020.

[43] BSI, *Mindeststandards des BSI zur Verwendung von Transport Layer Security (TLS).* https://www.bsi.bund.de/DE/Themen/StandardsKriterien/Mindeststandards_Bund/TLS-Protokoll/TLS-Protokoll_node.html. Zugriff 06.10.2020

[44] M. Freund, *freundma/GNUTLSwithFIDO2Extension.* https://github.com/freundma/GNUTLSwithFIDO2Extension. Git-Repository. Letzter Commit-Hash: 75c4fb6b81cdf0eadae1cb85affb69cdc7920667.

[45] T.-L. Breitkopf, *tom95br/tlslite-ng.* https://github.com/tom95br/tlslite-ng/tree/v1.0. Version v1.0. Git-Repository. Letzter Commit-Hash: 036174e41506c541b8cf256aa54dea9f78258971.

[46] N. Möller, *Nettle: a low-level cryptographic library.* Technisches Handbuch. 2017.

[47] *Copyright – License.* https://curl.haxx.se/docs/copyright.html. Zugriff am 03.11.2020.

[48] *Jansson.* https://digip.org/jansson/. Zugriff am 03.11.2020.

[49] *SQLite Is Public Domain.* https://www.sqlite.org/copyright.html. Zugriff am 03.11.2020.

[50] Free Software Foundation, Inc., *GNU General Public License 3.* https://www.gnu.org/licenses/gpl-3.0.html. Zugriff am 03.11.2020.

[51] S. Brandner, *Key words for use in RFCs to Indicate Requirement Levels.* IETF RFC 2119, 1997.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 16.12.2020 .....