Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

FIDO2 als TLS-1.3-Erweiterung

Bachelorarbeit

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

eingereicht von: Tom-Lukas Johann Breitkopf
geboren am:
geboren in:
Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Björn Scheuermann

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

Zusammenfassung

FIDO2 ist die aktuelle Standardsammlung der FIDO-Alliance und stellt eine kryptographisch sichere alternative zu Passwörtern dar [17]. Gegen viele Probleme bei der Verwendung von Passwörtern, wie Phishing- und Man-In-The-Middle-Attacken oder jene, die aus ihrer Mehrfachverwendung erwachsen [23], ist die FIDO2-Authentifizierung resistent [2]. Das Ziel dieser Arbeit ist es eine clientseitige Endpunktauthentifizierung nach den Vorgaben der FIDO2-Spezifikationen als Alternative zu bestehenden Mechanismen im TLS-1.3-Handshake zu ermöglichen. Dazu soll die FIDO2-Authentifizierung über den Erweiterungsmechanismus in den Handshake eingebunden werden. Neben einer Diskussion möglicher Kandidaten wird eine konkrete Erweiterung definiert. Der Fokus dieser liegt darauf, keine der Sicherheitseigenschaften des TLS-Protokolls oder der FIDO2-Spezifikationen durch die Verknüpfung beider einzuschränken. Sie orientiert sich in ihrem Aufbau und Ablauf am Handshake unter Verwendung von Clientzertifikaten. Eine Proof-Of-Concept-Implementation wird zur Verfügung gestellt, welche es erlaubt den Handshake unter Verwendung der Erweiterung nachzuvollziehen und zu testen. Mechanismen zur Registrierung von Nutzern sind kein Bestandteil der Ausarbeitungen.
Die Arbeit verdeutlicht insgesamt anhand einer theoretischen und praktischen Auseinandersetzung mit dem Thema, dass die sichere Ausführung von FIDO2-Authentifizierung über eine Erweiterung im TLS-1.3-Handshake möglich ist. Es wird gezeigt, dass dies nicht zwangsweise zu einer stark erhöhten Latenz führt. Durch die weite Verbreitung von TLS und seine wichtige Rolle bei der Sicherung von Client-Server-Kommunikation im Internet [51], bietet die Erweiterung die Möglichkeit, FIDO2-Authentifizierung für eine Vielzahl von Anwendungen mit einfachen Mitteln verfübgar zu machen. Insgesamt zeigt diese Arbeit dadurch auf, wie ein passwortloses Authentifizierungsverfahren für viele Menschen zugänglich und komfortabel nutzbar gemacht werden kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Akürzungsverzeichnis

AEAD Authenticated Encryption with Authenticated Data

API Application Programming Interface

CA Certificate Authority

CTAP Client to Authenticator Protocol

DoS Denial of Service

(EC)DHE (Eliplic Curve) Diffie-Hellman Ephemeral

FI FIDO2 with ID

FIDO Fast Identity Online

FN FIDO2 with Name

FTP File Transfer Protocol

GPL General Public License

HKDF HMAC-basierte KDF

HMAC Hash-basierter MAC

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IANA Internet Assigned Numbers Authority

IETF Internet Engineering Task Force

JSON JavaScript Object Notation

KDF Key Derivation Function

LGPL Lesser General Public License

MAC Message Authentication Code

NFC Near Field Communication

PKCS Public Key Credential Source

PoC Proof of Concept

PSK Preshared Key

RP Relying Party

RTT Roud Trip Time

SMTP Simple Mail Transfer Protocol

SRP Secure Remote Password

SSL Secure Socket Layer

TCC TLS 1.3 with Client Certificate

TCP Transmission Control Protocol

TFE TLS 1.3 with FIDO2 Extension

TLS Transport Layer Security. Früher SSL

U2F Universal Second Factor

UAF Universal Authentication Framework

VPN Virtual Private Network

WebAuthn W3C Web Authentication

1 Einleitung

Informationssysteme werden im öffentlichen Bereich für Verwaltung und Finanzen, in Unternehmen für Produktion und Logistik, im privaten, Mobilitäts- und Gesundheitsbereich eingesetzt [14]. Computer- und Netzwerktechnologie erlebt neben rapiden Entwicklungen eine stetig wachsende Nutzerzahl[1] [52]. Die Durchdringung aller Lebensbereiche mit Informationstechnik, lässt der Sicherung von IT-Infrastruktur eine große Bedeutung zukommen. Anwendungen finden bei Nutzern häufig nur dann Akzeptanz, wenn ein ausreichender Schutz der Daten gewährleistet ist. Unternehmen müssen Firmengeheimnisse und Kundendaten vor unautorisierten Zugriffen schützen [14].
Eine wichtige Rolle beim Schutz von Daten und Computersystemen kommt der Authentisierung und Authentifizierung eines Nutzers, einer Nutzerin oder allgemeiner einer Entität zu [30]. Die Authentisierung beschreibt in diesem Zusammenhang das Aufstellen einer Behauptung über eine partielle Identität. Die Authentifizierung besteht aus der Überprüfung dieser Behauptung. Eine partielle Identität wird charakterisiert durch eine Menge von Attributen, welche der Entität in dem bestimmten Kontext zugeordnet sind [25] – beispielsweise einem anwendungsspezifischen Pseudonym [26]. Erst nach erfolgreicher Prüfung der Behauptung sollten einer Entität sensible Daten zugänglich gemacht werden [30].
Eine gängige Form der Authentifizierung ist die Verwendung von Passwörtern [27]. Bereits im Jahr 2000 schrieb Bruce Schneider dazu: „Unfortunately the system of username and password works less well than people believe. The whole notion of passwords is based on an oxymoron. The idea is to have a random string that is easy to remember.“ [49]. Aus diesen widersprüchlichen Anforderungen erwachsen zahlreiche Probleme. Häufig wählen Nutzer schwache Passwörter, welche von Angreifern erraten oder über Brute-Force- und Dictionary-Attacken geknackt werden können. Wählen Nutzer komplexere Passwörter, speichern sie diese häufig, um sie sich besser merken zu können, an auch für Angreifer zugänglichen Orten oder verwenden das gleiche Passwort für verschiedene Systeme [23]. Diese Mehrfachverwendung führt wiederum eigene Probleme mit sich. Angenommen es gelingt einem Angreifer das Passwort für einen Onlinedienst mit geringen Sicherheitsstandards in Erfahrung zu bringen. Verwendet der Nutzer das Passwort ebenfalls für andere Dienste, ist es dem Angreifer möglich sich auch bei diesen illegitimen Zugang zu verschaffen – auch falls sie höhere Sicherheitsstandards erfüllen [27]. Die statische Natur eines Passwortes erlaubt dem Angreifer dabei nicht nur einmaligen, sondern wiederholten Zugang [23]. Auch unabhängig der Wahl des Passwortes bestehen Risiken bei dem Verfahren. In der Vergangenheit konnten Passwörter durch Angreifer mit Hilfe böswilliger Webseiten und E-Mails (Phishing) in Erfahrung gebracht werden, indem sie die unsichere

[1] Im Folgenden soll „Nutzer“ genau wie „Nutzerin“ auch für die jeweils andere und alle weiteren Geschlechteridentitäten stehen. Für eine bessere Lesbarkeit wird zumeist lediglich die maskuline Form angegeben. Dies gilt ebenfalls für andere geschlechtsspezifische Begriffe.

Speicherung der Passwörter auf dem Server ausnutzen [44] oder über Man-In-The-Middle- und Keyloggin-Attacken [23].
Es existieren jedoch auch Authentifizierungsmethoden, die nicht auf Passwörtern beruhen. So zum Beispiel auf Kryptographie basierende *Challenge-Response*-Verfahren. Seien A und B zwei Kommunikationspartner. Möchte sich A gegenüber B authentifizieren, so muss B eine unvorhersagbare Frage stellen – die Challenge. A kann diese nur unter Kenntnis von geheimem Wissen korrekt beantworten – in seiner Response. Die Umsetzung eines solchen Verfahrens kann über Einwegfunktionen erfolgen. Kennen sowohl A als auch B die zu verwendende Einwegfunktion und den Schlüssel, von dem sie abhängt, so kann eine Challenge von B aus einer Zufallszahl bestehen und die Response von A aus der Anwendung der Einwegfunktion auf die Challenge. B kann die von A empfangene Antwort überprüfen, indem er seinerseits die Einwegfunktion auf die Zufallszahl anwendet und mit der Antwort vergleicht. Wird anstelle dieses symmetrischen Verfahrens ein digitale Signatur verwendet, so kann die Authentifizierung stattfinden, ohne dass B das Geheimnis von A kennt [5]. Asymmetrische Verfahren bieten somit insbesondere den Vorteil, dass keine geheimen Informationen über einen Nutzer, wie dessen Passwort oder ein gemeinsamer Schlüssel, auf dem Server gespeichert werden müssen. Für den Nutzer entfällt bei Challenge-Response-Verfahren des Weiteren die Notwendigkeit, sich Passwörter merken zu müssen [27]. Stattdessen muss der private Schlüsselanteil des für asymmetrische Kryptographie benötigten Schlüsselpaares auf dem Gerät des Nutzers, auf einem mobilen Gerät oder einem Token gespeichert werden. Ein Token ist dabei ein physisches Gerät, welches mehrere Schlüssel generieren und speichern und notwendige Operationen für ein Challenge-Response-Verfahren ausführen kann [36].
Ein Challenge-Response-Verfahren kann anstatt eines Passwortes verwendet werden. Alternativ kann es als zweiter Faktor zur Passwort-Authentifizierung hinzugefügt werden [31]. Sei G das Gerät, welches den privaten Schlüssel des Nutzers speichert. Für eine erfolgreiche Anmeldung muss ein Nutzer das Wissen um das Passwort – den ersten Faktor – und den Besitzt des Gerätes G – des zweiten Faktors – nachweisen. Für einen Angreifer ist es dadurch auch nach Erbeuten des Passwortes nicht möglich, sich als legitimer Nutzer auszugeben, da er lediglich über einen der zwei benötigten Authentifizierungsfaktoren verfügt [28].

Trotz aller beschriebener Nachteile bleiben Passwörter noch heute die weit verbreiteteste Methode zur Nutherauthentifizierung [44] und Zweifaktorauthentifizierung wenig verbreitet [11]. Obwohl je nach Altersgruppe zwischen 55 und 79% aller für den „2019 State of Password and Authentication Security Behaviors Report“[39] befragten Personen angaben, in den letzten zwei Jahren mehr Wert auf die Privatsphäre und Sicherheit persönlicher Daten zu legen und 56% aller Befragten glaubten, durch die Verwendung eines Hardwaretokens ein besseres Sicherheitsniveau geboten zu bekommen, gaben lediglich 15% an, einen solchen als zweiten Faktor für persönliche Zwecke zu verwenden. Insgesamt nutzten ein Drittel der Befragten privat Zweifaktorauthentifizierung.

Die nachfolgende Arbeit befasst sich mit zwei Standards. Die Spezifikationen, welche unter dem Oberbegriff Fast Identity Online Version 2.0 (FIDO2) zusammen gefasst werden, beschreiben eine kryptographisch starke Nutzerauthentifizierung, welche auf einem asymmetrischen Challenge-Response-Verfahren beruht [19] und zur passwortlosen oder als Teil einer Mehrfaktor-Authentifizierung verwendet werden kann [20]. Das Transport Layer Security (TLS)-Protokoll stellt einen sicheren Kanal zwischen zwei Geräten her [41]. Es wird von allen bekannten Webbrowsern und Webservern unterstützt [32] und ist heute der de facto Standard zum Absichern von HTTP-Verbindungen [14].
Ziel der Arbeit ist es, die Durchführung clientseitiger FIDO2-Authentifizierung in TLS 1.3 über eine Erweiterung zu ermöglichen. Dadurch soll es für jeden Dienst, der TLS nutzt, möglich sein, FIDO2 durch wenige zusätzliche Parametern bei der Konfiguration von TLS anzubieten, ohne dabei FIDO2-Funktionalität implementieren zu müssen. Somit könnte für eine Vereinheitlichung in der Integration und Nutzung von FIDO2 gesorgt werden. Durch die enorm weite Verbreitung von TLS wäre jedoch insbesondere auch eine verstärkte Nutzung des kryptographiebasierten Verfahrens an Stelle von Passwörtern möglich.
Im Folgenden werden zunächst FIDO2 und das TLS-Protokoll genauer vorgestellt. Dabei werden die unterschiedlichen Schutzziele und die für den weiteren Verlauf der Arbeit notwendigen Mechanismen erläutert. Nach der Vorstellung der verwendeten Standards erfolgt die Beschreibung einer TLS-Erweiterung, welche die FIDO2-Authentifizierung im TLS-Verbindungsaufbau erlaubt. Dafür werden zunächst die theoretischen Vor- und Nachteile der Erweiterung unabhängig der Realisierung betrachtet, Anforderungen an die Umsetzung daraus abgeleitet und unterschiedliche Herangehensweisen für diese diskutiert. Anschließend wird der Vorschlag für die konkrete Erweiterung vorgestellt und die praktische Umsetzung anhand einer Proof-Of-Concept Implementation betrachtet.

2 Verwandte Arbeiten

In seiner Bachelorarbeit „Embedding of U2F into TLS 1.3“ [40] befasst sich Marco Reda mit der Integration des FIDO Universal Second Factor (U2F)-Standards in TLS 1.3. U2F erlaubt es Onlinediensten, die Sicherheit ihrer passwortbasierten Infrastruktur durch die Verwendung eines kryptographisch starken zweiten Faktors zu verbessern [50]. Reda diskutiert zwei verschiedene Herangehensweisen. Einerseits wird eine Integration ohne Änderungen am bestehenden Standard durch die geänderte Nutzung bereits vorhandener Felder beschrieben, andererseits eine Integration über die Verwendung neu definierter Erweiterungen. Um unerwartetes Verhalten des Protokolls zu verhindern, wird die Definition einer Erweiterung für den Transport von U2F-Nachrichten bevorzugt [40].
Auch Reda beschäftigt sich also mit der Thematik, einen kryptographisch sicheren Authentifizierungsmechanismus neben Clientzertifikaten in den TLS-1.3-Handshake zu integrieren. Diese Arbeit greift die Idee, den Erweiterungsmechanismus für diesen Zweck zu verwenden, auf und erweitert Redas Ausarbeitung um einen unabhängigen und konkreten Vorschlag für eine TLS-1.3-Erweiterung zur Umsetzung von FIDO-Authentifizierung. Anstatt des U2F-Standards wird die aktuelle Spezifikationssammlung der FIDO-Alliance FIDO2 verwendet. Diese ist abwärtskompatibel zu dem älteren FIDO-U2F-Standard [8] und kann nicht nur als zweiter, sondern auch als alleiniger Authentifizierungsfaktor verwendet werden [20]. Neben der theoretischen Betrachtung des Themas werden in dieser Arbeit darüber hinaus erstmals auch implementationsbedingte Eigenschaften einer solchen Erweiterung betrachtet und ihre praktische Umsetzbarkeit durch eine Proof-Of-Concept-Implementation erprobt.

3 Transport Layer Security

Das Transport Layer Security (TLS) Protokoll ist eins der weit verbreitetsten Sicherheitsprotokolle im Internet. Entwickelt wurde es 1994 von Netscape unter dem Namen Secure Socket Layer (SSL) [51] mit dem Ziel, eine einheitliche Lösung für die Sicherheitsprobleme bei der Kommunikation über das Internet zu finden [41]. Seit 1996 wird die Entwicklung unter dem Namen TLS von der Internet Engineering Task Force (IETF) fortgeführt [51]. Die Lage von TLS im Protokollstapel wird in Abbildung 3.1 gezeigt. Technisch gesehen befindet es sich in der Anwendungsschicht. Es kann von Programmiererinnen und Programmierern jedoch wie ein Transportschichtprotokoll verwendet werden [32]. Die Programmierschnittstelle von TLS wurde ähnlich zu der des von allen wichtigen Internetprotokollen verwendeten Transportschichtprotokolls TCP (Transmission Control Protocol) entworfen. Dies erlaubt es jedem Protokoll, welches über TCP übertragen werden kann, mit minimalen Änderungen auch über TLS verwendet zu werden [41]. Aus Sicht der Anwendungsschicht verhält sich TLS somit wie TCP mit zusätzlichen Sicherheitseigenschaften. TLS selbst bedient sich dabei der verbindungsorientierten und zuverlässigen Datenübertragung durch TCP [32]. Dass wegen des Schnittstellenentwurfs jedem Protokoll durch die Verwendung von TLS anstelle von TCP wichtige Sicherheitseigenschaften hinzugefügt werden können, ist einer der bedeutendsten Gründe für den Erfolg des Protokolls [41]. Besonders häufig wird es zur Absicherung der Kommunikation zwischen Webservern und -browsern über HTTP verwendet [32, 41]. Auf diese Weise geschützte Übertragungen werden HTTPS-Verbindungen genannt [14]. Fast alle bekannte Webserver unterstützen die Verwendung von TLS [32]. Das Sicherheitsprotokoll wird jedoch auch zum Schutz zahlreicher weiterer Internetprotokolle wie SMTP und FTP verwendet [41] und sichert somit auch den Transport von E-Mails sowie den Datentransfer zwischen Dateisystemen über das Internet ab [14, 32].
Diese Arbeit bezieht sich auf die aktuelle Version 1.3 [42] des Standards. In diesem Abschnitt werden neben den Schutzzielen des Protokolls der für die späteren Ausarbeitungen wichtige Erweiterungsmechanismus und Handshake beim Verbindungsaufbau erläutert.

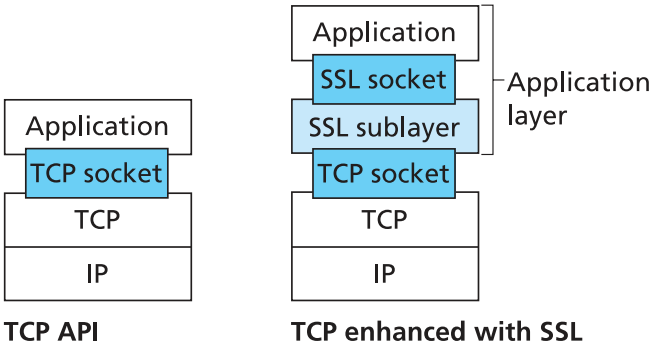

Abbildung 3.1: Einordnung von TLS (SSL) im Protokollstapel
Quelle: [32]

3.1 Schutzziele

Das TLS-Protokoll stellt einen sicheren Kanal zwischen zwei Kommunikationspartnern zur Verfügung [42]. Die Kommunikationspartner werden im Folgenden als *Server* – welcher einen Onlinedienst anbietet [37] – und *Client* – welcher einen Dienst in Anspruch nimmt [37] – bezeichnet. Ein *Endpunkt* beschreibt einen der Kommunikationspartner. Ein sicherer Kanal heißt in diesem Kontext, dass dieser die **Vertraulichkeit** und **Integrität** von Nachrichten sowie mindestens eine serverseitige **Endpunkt-Authentifizierung** gewährleistet [42].
Vertraulichkeit bedeutet, dass keine unauthorisierte Informationsgewinnung möglich ist [14]. Ein Angreifer darf also nicht in der Lage sein, den Klartext zu einer übertragenen Nachricht zu bestimmen [42]. Um die Vertraulichkeit einer Nachricht zu gewährleisten, wird diese verschlüsselt [41].
Erhaltene Nachrichtenintegrität garantiert, dass es dem Empfänger einer Nachricht möglich ist, zu überprüfen, ob diese bei der Übertragung verändert wurde. Dafür darf ein Angreifer keine legitime Nachricht durch eine gefälschte ersetzen können [48]. Um Nachrichtenintegrität sicher zu stellen, kann ein Message Authentication Code (MAC) verwendet werden [1].
Wird Endpunkt-Authentifizierung sichergestellt, so ist es dem Empfänger einer Nachricht möglich, die Herkunft dieser zu überprüfen. Es darf einem Angreifer also nicht möglich sein, sich als eine andere Entität auszugeben [48]. Durch serverseitige Endpunkt-Authentifizierung ist gewährleistet, dass der Server aus Sicht des Clients dem beabsichtigten Kommunikationspartner entspricht [41].
Die serverseitige Endpunkt-Authentifizierung wird in TLS 1.3 über Zertifikate umgesetzt. Vertraulichkeit und Integrität werden durch die Verwendung von Authenticated Encryption with Authenticated Data (AEAD)-Algorithmen sichergestellt. Alle drei Eigenschaften müssen für das TLS-Protokoll auch dann gelten, wenn ein Angreifer existiert, welcher vollständige Kontrolle über das Netzwerk besitzt [42].
Die clientseitige Endpunkt-Authentifizierung ist im TLS-Standard optional. Sie kann über ein Clientzertifikat erfolgen, welches vom Server explizit gefordert wird [42]. Die später eingeführte Authentifizierung über FIDO2 stellt eine Alternative zu Clientzertifikaten dar.

3.2 Erweiterungen

An verschiedene Nachrichten in TLS können so genannte Erweiterungen angehängt werden. Diese besitzen einen eindeutigen Erweiterungstypen und werden als Tag-Length-Value-Eintrag codiert. Die Liste der Erweiterungstypen wird von der Internet Assigned Numbers Authority (IANA) geführt [42].
Erweiterungen können von einem Client verwendet werden, um zusätzliche Funktionen von einem Server anzufordern. Dies geschieht in der ClientHello-Nachricht. Häufig bestehen Erweiterungen nicht nur aus einer Anfrage durch den Client, sondern ebenfalls aus einer Antwort durch den Server. Dafür übermittelt dieser in der ServerHello-, EncryptedExtensions-

oder Certificate-Nachricht eine Erweiterung gleichen Typs an den Client. Pro Erweiterungsblock darf dabei kein Erweiterungstyp mehrfach auftreten. Für jeden Erweiterungstypen muss darüber hinaus eindeutig definiert werden, in welchen Nachrichtentypen er auftreten darf. Auch dem Server ist es möglich, Anfragen an den Client zu senden. Diese werden an die CertificateRequest-Nachricht angehängt und vom Client mit einer passenden Erweiterung in der Certificate-Nachricht beantwortet [42].
Alle aufgeführten Nachrichtentypen werden im nächsten Teilabschnitt genauer erläutert. Auch Beispiele für die Verwendung von Erweiterungen werden an dieser Stelle genannt, da in TLS 1.3 Erweiterungen häufig notwendig sind, um neben dem Einführen neuer Funktionen auch die Kompatibilität zu älteren Protokollversionen aufrecht zu erhalten [42]. Der Erweiterungsmechanismus wird in Abschnitt 5 verwendet, um FIDO2-Authentifizierung in den TLS 1.3-Handshake einzubinden.

3.3 Der Handshake

TLS besteht aus zwei Hauptkomponenten: dem Handshake- und dem Record-Protokoll. Über das TLS-Handshake-Protokoll werden die Sicherheitsparameter einer Verbindung ausgehandelt. Das Record-Protokoll unterteilt die Daten in einzelne Records und verwendet die vom Handshake-Protokoll ausgehandelten Sicherheitsparameter, um die Records geschützt an den jeweiligen Kommunikationspartner zu übertragen [42]. Zur Verwendung von FIDO2 in TLS ist es notwendig, Änderungen am bestehenden Handshake vorzunehmen. Daher soll dieser im Folgenden erklärt werden. Am Record-Protokoll werden keine Änderungen vollzogen, weshalb es an dieser Stelle nicht näher eingeführt wird.
Abbildung 3.2 zeigt den exemplarischen Ablauf eines Handshakes zum Aufbau einer TLS-Verbindung. Erweiterungen werden durch gestrichelte Linien getrennt unterhalb der Nachricht dargestellt, in welcher sie übertragen werden. Der Handshake lässt sich in drei Phasen trennen: Schlüsselerzeugung, Festlegung weiterer Serverparameter und Authentifizierung [42].

Schlüsselerzeugung
In der ersten Phase werden Nachrichten ausgetauscht, um die Sicherheitsfähigkeiten von Client und Server zu bestimmen und Sitzungsschlüssel zu generieren, welche verwendet werden, um den restlichen Handshake und nachfolgende Daten zu schützen [42]. Der Client initiiert dafür den Nachrichtenaustausch mit einer *ClientHello*-Nachricht. Sie beinhaltet notwendige Parameter zur Schlüsselerzeugung. Diese bestehen zum einen aus einer Nonce, also zufällig generierten Bytes, welche mit in die Schlüsselgenerierung eingeht. Durch die Verwendung von Nonces werden in jedem Handshake unterschiedliche Schlüssel von der Key Derivation Function (KDF) erzeugt, auch falls bereits etablierte geheime Informationen wiederverwendet werden. Zum anderen werden vom Client unterstützte Cipher Suites übertragen, also Paare aus AEAD-Algorithmen und Hash-Algorithmen, welche für die HMAC-basierte KDFs (HKDFs) verwendet werden können. Weitere Bestandteile der

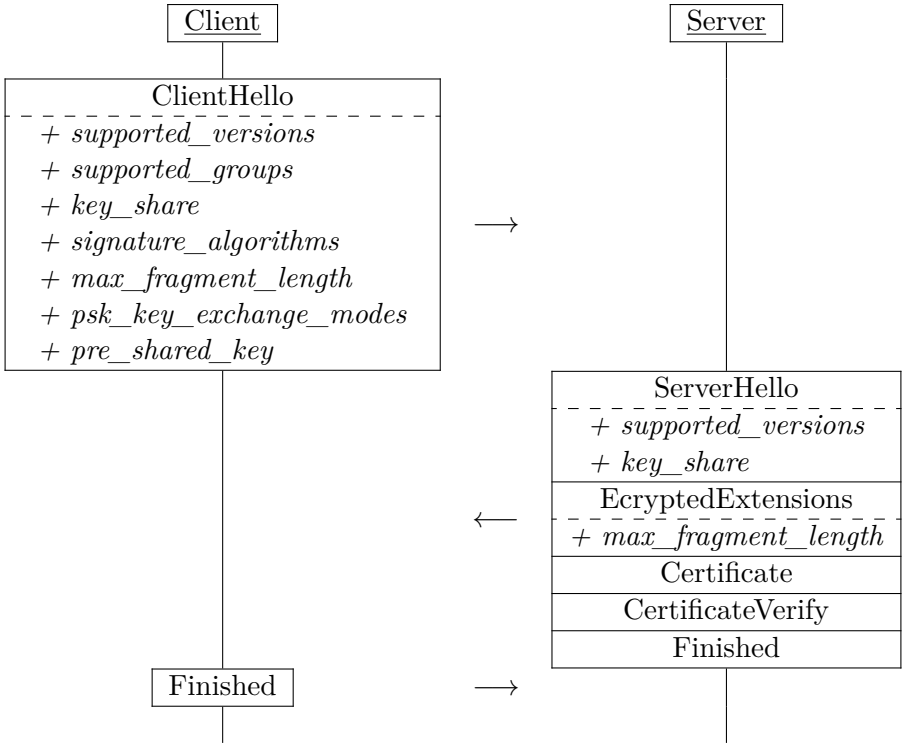

Abbildung 3.2: Der TLS 1.3 Handshake
Quelle: Eigene Abbildung nach [42]

ClientHello-Nachricht müssen aus Kompatibilitätsgründen zu alten Protokollversionen übertragen werden, sind für den TLS-1.3-Handshake jedoch nicht von Bedeutung. Andere Informationen wiederum müssen aus dem gleichen Grund in Erweiterungen ausgelagert werden. Die „supported_versions“-Erweiterung muss in TLS 1.3 übertragen werden, um die Protokollversion zwischen Client und Server auszuhandeln. In der „supported_groups“-Erweiterung werden vom Client unterstützte (EC)DHE-Gruppen für die Diffie-Hellman-Schlüsselgenerierung übertragen und in „key_share“ Schlüsselanteile für diese oder einen Teil dieser Gruppen. Zusätzlich ist es möglich, Identitäten bereits vereinbarter Schlüssel (Preshared Keys (PSKs)) in „pre_shared_key“ zu übergeben. In diesem Fall müssen über „psk_key_exchange_modes“ ebenfalls die unterstützen Modi übermittelt werden, also inwiefern ein PSK in Kombination mit (EC)DHE-Schlüsselgenerierung verwendet werden kann. Wird kein PSK verwendet, müssen in der „signature_algorithms“-Erweiterung unterstütze Algorithmen für digitale Signaturen angegeben werden, welche in späteren Nachrichten zur Authentifizierung des Servers verwendet werden dürfen [42].
Der Server verarbeitet die vom Client empfangenen Informationen, um die kryptographischen Parameter der Verbindung festzulegen. Sollte die Anfrage des Clients nicht kompatibel sein mit den Sicherheitsanforderungen des Servers, zum Beispiel weil es keine Überschneidung der von beiden unterstützen (EC)DHE-Gruppen gibt, bricht der Server den Handshake wegen mangelnder Sicherheit ab. Andernfalls überträgt er in seiner *ServerHello*-Nachricht ebenfalls eine Nonce und die ausgewählte Cipher Suite aus der vom Client übertragenen Liste. In der „supported_versions“-Erweiterung gibt der Server an, welche

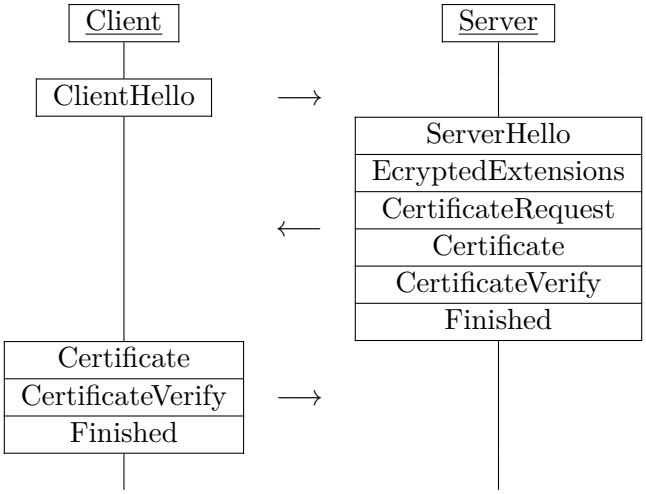

Abbildung 3.3: Der TLS 1.3 Handshake mit Clientzertifikat
Quelle: Eigene Abbildung nach [42]

der angebotenen TLS-Versionen verwendet wird. In Erweiterungen übermittelt er seinen Schlüsselanteil für eine ausgewählte (EC)DHE-Gruppe, die gewählte PSK-Identität oder beides an den Client. Im Beispiel-Handshake in Abbildung 3.2 überträgt der Server einen neu erzeugten Schlüsselanteil. Anhand der ClientHello- und ServerHello-Nachricht können Client und Server das gemeinsame Schlüsselmaterial berechnen. Die Schlüsselgenerierungsphase ist somit nach Übertragen der ServerHello-Nachricht abgeschlossen. Alle Nachrichten nach dem ServerHello werden verschlüsselt übermittelt. Falls neue (EC)DHE-Schlüssel für jede Verbindung generiert und nach Ende der Verbindung gelöscht werden oder ein PSK in Kombination mit (EC)DHE-Schlüsselgenerierung verwendet wird, so ist die *Forward Secrecy* der mit den Schlüsseln geschützten Daten sicher gestellt. Das bedeutet, dass die Kompromittierung von Langzeitschlüsseln keine Entschlüsselung der übertragenen Daten ermöglicht [42].

Serverparameter

In der zweiten Phase des Handshakes werden weitere Serverparameter festgelegt. Unter anderem wird an dieser Stelle bestimmt, ob eine clientseitige Authentifizierung stattfinden soll. Die versandten Nachrichten werden mit den in der vorherigen Phase generierten Schlüsseln geschützt. Der Server verschickt in der Phase eine oder zwei Nachrichten. Als erstes muss die *EncryptedExtensions*-Nachricht verschickt werden. Diese beinhaltet Antworten auf mögliche Client-Erweiterungen, welche nicht benötigt wurden, um die kryptographischen Parameter der Verbindung festzulegen [42]. So kann der Server an dieser Stelle zum Beispiel seine Antwort auf die „max_fragment_lenght“-Erweiterung übertragen, um die Festlegung einer kleineren maximalen Länge der Klartextnachrichten zu bestätigen [13, 42]. Die zweite Nachricht, welche der Server in dieser Phase optional verschickt, ist die *CertificateRequest*-Nachricht. Sie zeigt an, dass eine clientseitige Authentifizierung über Zertifikate vom Server verlangt wird. Wünscht der Server diese nicht, so wird keine

CertificateRequest-Nachricht verschickt [42]. Abbildung 3.3 zeigt einen TLS-Handshake mit zertifikatbasierter Clientauthentifizierung. Für eine simplere Darstellung werden keine Erweiterungen gezeigt.

Authentifizierung

In der letzten Phase des Handshakes findet die serverseitige und optional auch die clientseitige Authentifizierung statt. Darüber hinaus wird das generierte Schlüsselmaterial bestätigt und die Integrität des Handshakes überprüft. Dafür sind drei Nachrichten notwendig, welche zunächst der Server versendet. In der *Certificate*-Nachricht ist das Zertifikat des Servers enthalten. Die *CertificateVerify*-Nachricht beinhaltet eine Signatur über den bisherigen Handshake mit dem passenden privaten Schlüssel zu dem öffentlichen Schlüssel im vorher versandten Zertifikat. Dadurch wird explizit bewiesen, dass der Endpunkt den privaten Schlüssel zu dem Zertifikat besitzt. In der Certificate und CertificateVerify-Nachricht dürfen nur Signaturverfahren verwendet werden, welche der Client laut seiner „signature_algorithms“-Erweiterung unterstützt. Wird ein PSK verwendet, so findet über diesen die Authentifizierung statt und die Certificate- und CertificateVerify-Nachricht werden nicht verschickt. Die letzte Nachricht des Servers ist die *Finished*-Nachricht. Sie beinhaltet einen MAC über den gesamten Handshake. Dadurch wird die Integrität des Handshakes sicher gestellt, die generierten Schlüssel werden bestätigt und an die Identität des Endpunktes gebunden [42].
Nach dem Empfangen und Auswerten der vom Server erhaltenen Authentifizierungsnachrichten antwortet der Client. Wurde vom Server clientseitige Authentifizierung angefordert, so versendet der Client, wie in Abbildung 3.3 zu sehen, eine *Certificate*- und eine *CertificateVerify*-Nachricht. Als letzte Nachricht des Handshakes übermittelt auch der Client eine *Finished*-Nachricht [42].

Weitere Erweiterungen und Nachrichten, welche in den vorangegangenen Ausführungen keine Erwähnung fanden, können im TLS-1.3-Handshake auftreten. Einfachheitshalber wurden an dieser Stelle lediglich Varianten des Verbindungsaufbaus betrachtet, welche für den weiteren Verlauf der Arbeit von Relevanz sind.

4 Fast Identity Online Version 2.0

Die Fast Identity Online (FIDO) Alliance wurde 2012 von PayPal, Lenovo, Nok Nok Labs, Validity Sensors, Infineon und Agnitio gegründet [18]. Heute zählen unter anderem Google, Amazon, Microsoft, Alibaba, Intel, Samsung und Facebook zu ihren Mitgliedern [16]. Ziel der Allianz ist es, die übermäßige Abhängigkeit von Passwörtern zu reduzieren. Dafür sollen offene Standards für Authentifizierungsmethoden geschaffen werden, welche nicht nur sicherer als klassische Passwortverfahren, sondern darüber hinaus einfacher für Nutzer anzuwenden und für Servicebetreiber zu verwalten sind [15].
Die ersten Standards der FIDO Alliance wurdem im Jahr 2014 mit FIDO Universal Authentication Framework (UAF) für passwortlose und Universal Second Factor (U2F) für Zweifaktorauthentifizierung vorgestellt [18].
Die Authentifizierungsmethode aller FIDO-Standards basiert auf asymmetrischer Kryptographie. Registriert sich ein Nutzer bei einem Onlinedienst, so erzeugt das verwendete oder ein verbundenes Gerät ein neues Schlüsselpaar. Es sichert den privaten Schlüssel und registriert den öffentlichen Schlüssel bei dem Onlinedienst. Bei der Authentifizierung signiert das Gerät, nach Freigabe des Nutzers, eine vom Onlinedienst gestellte Challenge und weist somit den Besitz des passenden privaten Schlüssels zu dem öffentlichen Schlüssel nach [19]. Die Komponente, welche das Schlüsselpaar generiert und die kryptographische Signatur ausführt, wird im Folgenden *Authentifikator* genannt. Der Onlinedienst, welcher Nutzer registriert und authentifiziert, wird als *Relying Party* (RP) bezeichnet [4].
Authentifizierungsverfahren werden in drei Kategorien unterschieden, je nach dem, ob sie auf der Kenntnis spezifischen Wissens, dem Besitz eines Gerätes oder biometrischen Daten basieren [1]. Die drei Kategorien werden häufig auch als „something you know“, „something you have“ und „something you are“ bezeichnet [11]. FIDO-Authentifikatoren fallen immer mindestens in die zweite Kategorie, da sie den Besitz eines Gerätes, respektive

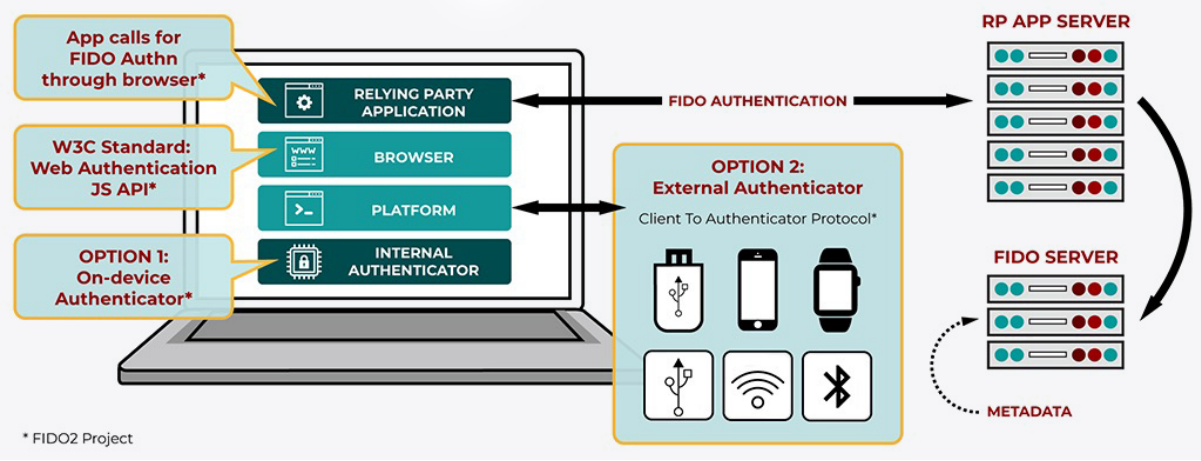

Abbildung 4.1: FIDO2 im Überblick
Quelle: [20]

des darauf gespeicherten privaten Schlüssels, nachweisen. Sie können, wie später erläutert, jedoch unter Umständen auch weitere Kategorien abdecken [3, 4].
Die aktuellste Standardsammlung der FIDO Alliance heißt FIDO2. Sie besteht aus den Spezifikationen von W3C Web Authentication (WebAuthn) und dem Client to Authenticator Protocol (CTAP) [17] und ist abwärtskompatibel zu U2F [8]. FIDO2 unterstützt passwortlose und Mehrfaktorauthentifizierung [20]. Es können interne Authentifikatoren, welche sich direkt auf dem verwendeten Gerät befinden, oder externe Authentifikatoren, wie Sicherheitstokens oder mobile Geräte, verwendet werden [22]. Die einzelnen Komponenten von FIDO2 und deren Zusammenspiel werden in Abbildung 4.1 gezeigt.
WebAuthn definiert eine API, welche es Browsern und Plattformen ermöglicht, FIDO-Authentifizierung zu unterstützen [20]. Seit März 2019 ist WebAuthn offizieller Webstandard [17]. CTAP ermöglicht die Kommunikation zwischen der Plattform eines Nutzers und externen Authentifikatoren. Es besteht aus den Protokollen CTAP1 und CTAP2 [8]. CTAP1 ist der neue Name des U2F-Protokolls und erlaubt es, Geräte, welche lediglich den alten Standard unterstützen, auch über FIDO2 zu nutzen. CTAP2 kontrolliert den Nachrichtenaustausch mit FIDO2-kompatiblen Geräten und ermöglicht diesen, neue FIDO2-Funktionen wie passwortlose Authentifizierung zu verwenden. Sowohl CTAP1 als auch CTAP2 können über USB, NFC oder Bluetooth verwendet werden [8, 20].
Bei der Ausführung von FIDO2 als TLS-Erweiterung werden WebAuthn-Informationen im TLS-Handshake übertragen. CTAP-Nachrichten werden nach wie vor zwischen Client-Plattform und externen Authentifikatoren ausgetauscht. Die nachfolgenden Erläuterungen konzentrieren sich daher auf WebAuthn und die für die Erweiterung relevanten Operationen.

4.1 Schutzziele

Ein Hauptziel aller Authentifizierungsverfahren der FIDO-Alliance ist es, die Sicherheit gegenüber passwortbasierten Methoden zu erhöhen [15]. Um dieses Ziel zu erreichen, werden verschiedene Ansprüchen an die FIDO2-Spezifikationen gestellt und unterschiedliche Maßnahmen ergriffen, diese umzusetzen.
Wichtiger Bestandteil einer sicheren Authentifizierung ist die kryptographische Stärke der verwendeten Operationen. Dazu existiert eine Liste moderner und starker Kryptoprimitiven, welcher sich FIDO-zertifizierte Authentifikatoren bedienen dürfen. Außerdem ist es Relying Parties möglich, über den FIDO Metadata Service den Status verschiedener Authentifikatoren abzufragen. Auf diese Weise können sie über die Kompromittierung bestimmter Modelle benachrichtigt werden und Informationen über die Sicherheitseigenschaften eines Authentifikators berücksichtigen [2].
Ein weiteres Ziel der FIDO2-Authentifizierung ist es, so wenig Informationen wie möglich an Relying Parties zu übertragen, welche genutzt werden können, um einen Nutzer zu identifizieren. Für jede Relying Party wird daher von dem Authentifikator ein neues Schlüsselpaar generiert, welches ausschließlich für diese verwendet werden kann. Dadurch ist es zwei Relying Parties, welche mit dem gleichen Nutzer kommunizieren, nicht möglich,

anhand des verwendeten Schlüsselpaares die zwei Konversationen dem einen Nutzer zuzuordnen. Da jede Relying Party lediglich öffentliche oder verschlüsselte Informationen über ein Schlüsselpaar speichert, welches ihr allein zugeordnet ist, ist es einer Relying Party unmöglich, Informationen preiszugeben, welche einem Angreifer dabei helfen könnten, sich als ein bestimmter Nutzer gegenüber einer anderen Relying Party auszugeben. Zwei Authentifikatoren können nicht unterschieden werden, solange genügend (> 100.000) Authentifikatoren mit dem gleichen Attestierungsschlüssel hergestellt wurden [2]. Dessen Funktion wird in Abschnitt 4.2 erläutert.
Alle privaten Schlüssel sind auf dem Authentifikator geschützt. Nur durch eine explizite Nutzerinteraktion können sie verwendet werden [2]. Für diese Nutzeraktion gibt es zwei verschiedene Möglichkeiten. Sie kann aus einem Test auf Anwesenheit des Nutzers bestehen, für welchen dieser den Authentifikator meist lediglich berühren muss. Dadurch wird sichergestellt, dass *ein* Nutzer beabsichtigt die angezeigte Operation auszuführen. *Welcher* Nutzer die Aktion frei gibt, wird jedoch nicht verifiziert. Deshalb existiert die zweite Möglichkeit. Bei dieser findet eine Nutzerverifikation statt. Sie kann zum Beispiel aus der Eingabe eines Passwortes oder einer PIN oder aus der Verwendung eines biometrischen Sensors bestehen. Der Authentifikator deckt über den Mechanismus der Nutzerverifikation nicht nur die „something you have“-Kategorie der Authentifizierung ab, sonder je nach Verifikationstyp auch eine der anderen beiden Kategorien. Wird eine PIN verwendet, so wird ebenfalls „something you know“ überprüft, bei einem biometrischen Sensor „something you are“. Die Verifikation findet lokal gegenüber dem Authentifikator statt, um eine Operation freizugeben, und nicht gegenüber der Relying Party. Diese erfährt lediglich das Ergebnis der Verifikation [4].
Die FIDO2-Authentifizierung ist resistent gegenüber Phishing-Angriffen. Angenommen eine betrügerische Webseite A bringt einen Nutzer dazu, sich ihr gegenüber zu authentifizieren, um Anmeldedaten für eine andere Webseite B in Erfahrung zu bringen. Entweder stellt der Authentifikator fest, dass für A kein Schlüsselpaar existiert, und bricht die Authentifizierung ab oder er verwendet das Schlüsselpaar, welches eigens für A generiert wurde. Da dies ein anderes ist als für Webseite B, können die an A übermittelten Informationen nicht genutzt werden, um sich gegenüber B zu authentifizieren. Sollte ein Angreifer über einen Phishing-Angriff die PIN eines Nutzers für die Nutzerverifikation in Erfahrung bringen, so benötigt er dennoch den Authentifikator, um sich erfolgreich zu authentifizieren. Da die Nutzerverifikation lokal gegenüber dem Authentifikator stattfindet und die PIN dem RP-Server nicht bekannt ist, reicht sie nicht aus um sich gegenüber diesem zu authentifizieren. Gleiches gilt auch für andere Verifikationsmethoden [2].
Es existieren weitere Schutzziele und verschiedene Maßnahmen diese umzusetzen in FIDO2. So können zum Beispiel über einen monoton steigenden Signaturzähler, welchen Authentifikatoren an Relying Parties übermitteln, duplizierte Authentifikatoren entdeckt werden. Andere Mechanismen verhindern Replay-, Forwarding-, DoS- und Impersonation-Angriffe [2]. Eine vollständige Übersicht ist in [2] zu finden.

4.2 Registrierung

Die Registrierung von Nutzern wird in der TLS-Erweiterung nicht berücksichtigt. Um den Authentifizierungsprozess nachvollziehen zu können, ist es dennoch wichtig zu verstehen, welche Schritte bei der Registrierung unternommen werden. Ihr Ablauf soll daher im Folgenden erläutert werden. Nicht alle optionalen Bestandteile und Details des Prozesses werden jedoch berücksichtigt. Abbildung 4.2 verdeutlicht den Hergang der Registrierung unter Verwendung der WebAuthn API, welche im Folgenden in die Schritte „Aufforderung durch den Server“, „Generierung der PKCS“ und „Antwort des Authentifikators“ aufgeteilt wird.

Aufforderung durch den Server
Nach der initialen Kontaktaufnahme durch die Anwendung fordert der Server den Authentifikator dazu auf, eine neue Public Key Credential Source (PKCS) zu generieren. PKCSs sind kritisch für die Funktionsweise von FIDO2. Sie werden für die Erzeugung der Signatur in jedem späteren Authentifizierungsprozessen benötigt [4].
Die Parameter der Aufforderung durch den Server werden zunächst in Schritt 1 als „PublicKeyCredentialCreationOptions“ an die JavaScript-Applikation übermittelt. Zusammen mit einer vom Server generierten Challenge werden Informationen über den Nutzer und über die Relying Party übertragen. Die Nutzerinformationen beinhalten die ID des Nutzers, seinen Nutzernamen und einen Anzeigenamen. Die RP-Informationen umfassen den Namen der Relying Party und optional deren ID. Ist letztere nicht vorhanden, so wird ihr Wert auf die mit der JavaScript-Applikation assoziierten Domain gesetzt. Andernfalls muss der Wert ein registrierbarer Domain-Suffix oder gleich (siehe 7.5.1 in [24]) ebenjener Domain sein [4]. Die RP-ID ist von Bedeutung, da sie, wie später erläutert, den Gültigkeitsbereich der PKCS einschränkt. Auch Anforderungen an die Eigenschaften der PKCS in den „pubKeyCredParams“ und optional weitere Parameter werden übertragen [4]. Die vollständige Parameterliste kann in Abschnitt 5.4 von [4] nachgelesen werden. Die JavaScript-Anwendung gibt die vom Server empfangenen Daten über die `create()`-Methode der WebAuthn API an den Browser weiter [34].
In Schritt 2 überträgt der Browser die Aufforderung zur Generierung einer PKCS an den Authentifikator [34]. Dafür wird dessen `authenticatorMakeCredential()`-Methode verwendet. Der Browser generiert die „clientData“. Diese bestehen aus dem Typ der Operation, also dem String „webauthn.create“, der vom Server gestellten Challenge in Base64url-Encoding und einer Origin. Die Origin ist an das Objekt gebunden, welches die WebAuthn API zur Registrierung eines Nutzers aufgerufen hat, und beinhaltet insbesondere die damit assoziierte Domain. Der SHA256-Hash der JSON-serialisierten Form dieser „clientData“ wird zusammen mit den Informationen über Nutzer und Relying Party sowie den „pubKeyCredParams“ und weiteren optionalen Parameter des Servers an den Authentifikator weitergegeben [4]. Die Liste möglicher Argumente ist in Abschnitt 5.1 von [8] zu finden.

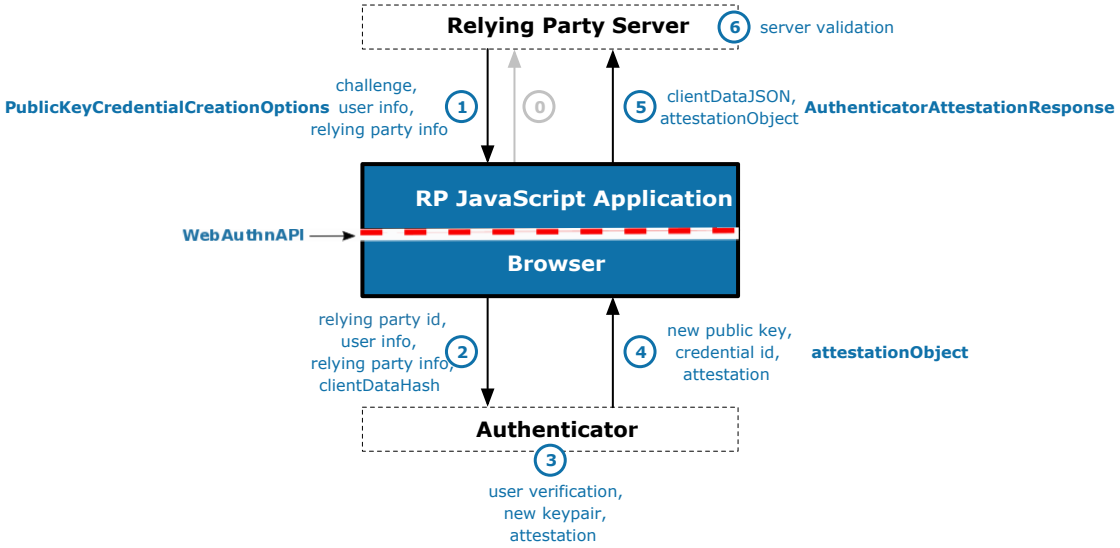

Abbildung 4.2: Ablauf der WebAuthn Registrierung
Quelle: [4]

Generierung der PKCS

In Schritt 3 wird eine neu PKCS vom Authentifikator generiert. Dafür muss der Nutzer zunächst über eine Interaktion mit diesem sein Einverständnis geben. Bei der Generierung jeder PKCS wird ein neues Schlüsselpaar erzeugt. Der dafür zu verwendende Algorithmus wird den „pubKeyCredParams“ entnommen [8]. Für die Speicherung einer PKCS auf einem Authentifikator bestehen zwei Möglichkeiten. Sie kann im persistenten Speicher des Authentifikators, der Clientanwendung oder des Clientgerätes gehalten werden. Auf diese Weise gesicherte PKCS werden als *resident* PKCS bezeichnet. Alternativ kann der private Schlüsselanteil so verschlüsselt werden, dass nur der Authentifikator ihn wieder entschlüsseln kann. Der generierte Ciphertext wird als ID der PKCS gewählt, welche von der Relying Party gespeichert wird. Dadurch können unendlich viele private Schlüssel auf einem Authentifikator verwendet werden. Bevor dieser in einem späteren Authentifizierungsvorgang eine PKCS mit auf diese Weise gesichertem privatem Schlüssel verwenden kann, muss er jedoch zunächst deren ID von der Relying Party empfangen [4].
Neben dem privaten Schlüsselanteil werden weitere Informationen gespeichert. Insgesamt wird eine PKCS definiert durch ihren Typ, eine Credential-ID, die RP-ID, den privaten Schlüsselanteil sowie optional die ID des Nutzers und dessen Anzeigenamen. Der Typ muss nach aktuellen Vorgaben auf „public-key“gesetzt sein. Die Credential-ID dient als eindeutiger Bezeichner einer PKCS. Die Nutzer-ID wird im Falle eines resident PKCS verwendet, um einen Nutzer gegenüber einer Relying Party gleichzeitig zu identifizieren und authentifizieren. Die RP-ID legt den Gültigkeitsbereich (Scope) der PKCS fest. Die Origin eines Objekts, welches über die WebAuthn API eine Authentifizierung anfordert, setzt sich zusammen aus einem Schema, einem Port, dem Host und der Domain. Die effektive Domain lässt sich aus Host und Domain bestimmen [24]. Eine PKCS kann für eine gegebene Origin verwendet werden, falls die RP-ID der PKCS gleich der effektiven

Domain der Origin oder ein registrierbarer Domain-Suffix dieser ist und das Schema der Origin „https“ lautet. Für den Port gibt es keine Vorschriften.
Die neu generierte PKCS wird in allen folgenden Authentifizierungsprozessen benötigt, um eine kryptographische Signatur zu erzeugen und somit den Besitzt des privaten Schlüsselanteils zu dem auf dem Server gesicherten öffentlichen Schlüsselanteil zu beweisen [4].

Antwort des Authentifikators

Nach Generierung der neuen PKCS liefert der Authentifikator in Schritt 4 ein „attestationObject“ an den Browser zurück [34]. Dieses Objekt enthält die „authenticatorData“ und ein „attestationStatement“. Letzteres beinhaltet eine Signatur über die „authenticatorData“ und den Hash der „clientData“ – also insbesondere auch die vom Server gestellte Challenge. Für die Signatur wird der private Attestierungsschlüssel verwendet. Dieser kann zum Beispiel vom Hersteller bei der Produktion fest auf alle Authentifikatoren des gleichen Modells geschrieben werden. Der öffentliche Anteil dieses Schlüssels wird im Attestierungszertifikat mitgeliefert. Über den Attestierungsschlüssel werden Eigenschaften des Authentifikators und dessen Hersteller bestätigt. Neben der Nutzung von Zertifikaten sind weitere Typen der Attestierung möglich, wie die Selbstattestierung. Damit das attestationStatement korrekt interpretiert werden kann, enthält es einen Bezeichner des Formats [4]. Die vollständige Übersicht der Typen und Formate ist in 6.4.2 und 6.4.3 von [4] aufgeführt.
Die „authenticatorData“ beinhalten Informationen über die neu generierte PKCS und den Authentifikator, welcher sie generiert hat. Insbesondere ist in ihnen die eindeutige ID der PKCS und ihr öffentlichen Schlüsselanteil enthalten sowie die Kennung (AAGUID), welche das Model des dazugehörigen Authentifikators beschreibt [4]. Auf weitere Bestandteile wird erst im Abschnitt zur Authentifizierung genauer eingegangen.
Über den Rückgabewert des `create()`-Aufrufs übergibt der Browser das „attestationObject“ und die JSON-serialisierte Form der zuvor generierten „clientData“ an die JavaScript-Anwendung. In Schritt 5 überträgt die Anwendung diese Informationen als „AuthenticatorAttestationResponse“ an den RP-Server [34].
Zuletzt verifiziert der RP-Server die Antwort des Authentifikators in Schritt 6. Dafür prüft er, dass Typ, Challenge und Origin der „clientData“ mit den erwarteten Werten übereinstimmen. Auch die Bestandteile der „authenticatorData“ werden überprüft. Anhand des Formatbezeichners kann das „attestationStatement“ interpretiert und die Signatur geprüft werden. Anschließend wird die Vertrauenswürdigkeit der Attestierung eingeschätzt. Dafür wird, falls vorhanden, die mitgelieferte Zertifikatskette überprüft und der Typ der Attestierung mit der Sicherheitspolice des RP-Servers abgeglichen. War die Verifikation erfolgreich, die Attestierung wurde als vertrauenswürdig eingestuft und die PKCS-ID wurde vorher noch nicht für einen anderen Nutzer verwendet, so wird dem Nutzer die PKCS zugeordnet, öffentliche Informationen dieser gespeichert und die Registrierung erfolgreich beendet [4].

4.3 Authentifizierung

Die zweite wichtige Operation, für welche die WebAuthn API neben der Registrierung verwendet wird, ist die Authentifizierung. Bei diesem Vorgang wird gegenüber einer Relying Party kryptographisch bewiesen, dass der verwendete Authentifikator den privaten Schlüssel einer zuvor registrierten PKCS kontrolliert und der Nutzer der Authentifizierung zugestimmt hat [4]. Sie folgt einem ähnlichen Ablauf wie die Registrierung und wird in Abbildung 4.3 dargestellt. Die Authentifizierung wird im Folgenden in die Teilschritte „Anfrage des Server“, „Generierung einer Assertion“ und „Antwort des Authentifikators“ unterteilt.

Anfrage des Servers
Die mit 0 gekennzeichnete initiale Anfrage zu Beginn der Zeremonie wird in WebAuthn nicht spezifiziert [34]. In Abschnitt 5 dieser Arbeit wird eine Möglichkeit sie umzusetzen gezeigt. Nach der initialen Kontaktaufnahme fordert der RP-Server den Client auf, eine Challenge durch eine Signatur mit einer bereits registrierten PKCS zu beantworten [4]. Dafür versendet er in Schritt 1 die Challenge an die RP-Anwendung. Diese ist Teil der „PublicKeyCredentialRequestOptions“. Sie muss vom RP-Server zufällig generiert werden und eine Länge von mindestens 16 Bytes besitzen. Weitere optionale Bestandteile sind die ID der Relying Party, falls wie zuvor bei der Registrierung nicht ein Standardwert verwendet werden soll, und eine Liste der von der Relying Party für einen gegebenen Nutzer akzeptierten PKCSs, welche in „allowCredentials“ anhand ihrer ID identifiziert werden. Ein Timeout-Wert, der die maximale Zeit angibt, welche die RP bereit ist auf eine Antwort zu warten, sowie eine Liste von Erweiterungen können ebenfalls übergeben werden. Ein weiterer wichtiger Parameter beschreibt die Anforderungen der Relying Party bezüglich der Nutzerverifikation. Standardmäßig ist dieser Wert auf „preferred“ gesetzt, was bedeutet, dass Nutzerverifikation bevorzugt wird, sollte der Authentifikator sie unterstützen, die Authentifizierung jedoch nicht fehlschlägt, sollte keine Verifikation des Nutzers erfolgen. Weitere Möglichkeiten für diesen Wert sind „required“ und „discouraged“ [4]. Durch einen Aufruf der `get()`-Methode der WebAuthn API werden die Parameter im Regelfall ohne oder mit geringen Änderungen an den Browser übergeben [34].
In Schritt 2 werden für die Authentifizierung notwendige Parameter vom Browser an den Authentifikator weitergeleitet. Zunächst muss er dafür überprüfen, dass die empfangene RP-ID der effektiven Domain der Origin des aufrufenden Objektes oder einem registrierbaren Domain-Suffix dieser entspricht und mögliche Erweiterungen verarbeiten. Ähnlich zur Registrierung generiert er anschließend die „clientData“. Der Typ entspricht in diesem Fall dem String „webauthn.get“, die Challenge besteht erneut aus der Base64url-codierten Challenge des RP-Servers und die Origin aus der des aufrufenden Objekts. Optional können Informationen des Token-Binding-Protokolls enthalten sein. Die „clientData“ werden JSON-serialisiert und ein SHA256-Hash darüber generiert [4]. Dieser wird zusammen mit der RP-ID, den optionalen „allowCredentials“ und Flags, welche angeben ob Nutzerpräsenz oder Nutzerverifikation erforderlich ist, an die `authenticatorGetAssertion()`-Methode

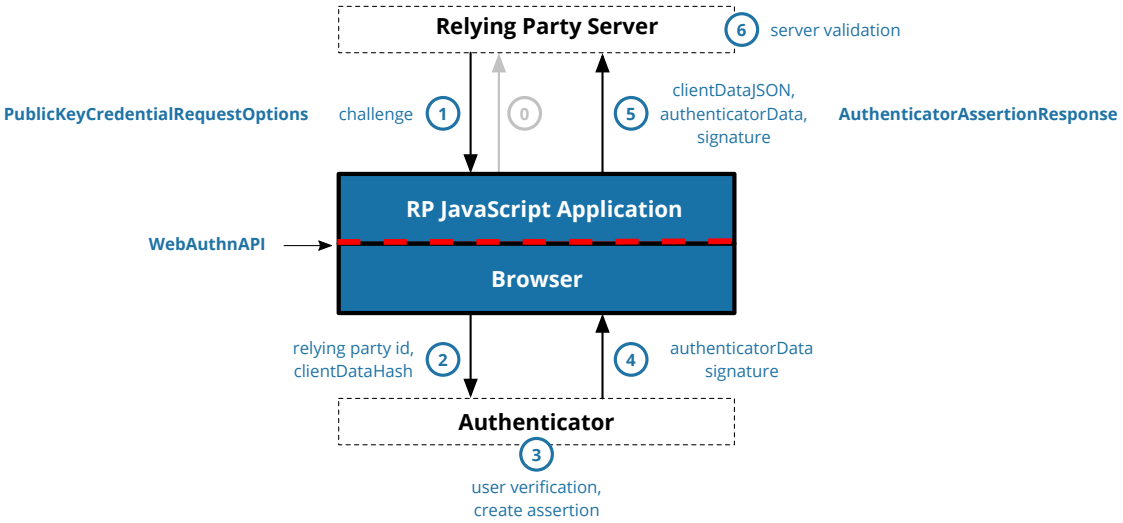

Abbildung 4.3: Ablauf der WebAuthn Authentifizierung
Quelle: [4]

geeigneter Authentifikatoren weitergeleitet [8]. Die vollständige Liste der Parameter ist in 5.2 von [8] aufgeführt. Ein geeigneter Authentifikator besitzt eine PKCS, welche an die gegebene RP-ID gebunden ist. Nur die Elemente der „allowCredentials“-Liste, welche eine auf dem entsprechenden Authentifikator vorhandene PKCS beschreiben, werden übergeben. Die Liste kann darüber hinaus Hinweise enthalten, auf welche Weise mit dem Authentifikator kommuniziert werden kann – zum Beispiel per USB oder NFC [4].

Generierung einer Assertion

Eine Assertion ist das Ergebnis der `authenticatorGetAssertion()`-Methode. Ihre Verifikation erlaubt es dem RP-Server, die Authentifizierung abzuschließen [4]. Um in Schritt 3 eine Assertion zu generieren, muss der Authentifikator zunächst eine passende PKCS auswählen. Falls keine „allowCredentials“-Liste an den Authentifikator übergeben wurde, wählt er alle resident PKCSs aus, welche an die angegebene RP-ID gebunden sind. Falls sie vorhanden ist, wählt er alle an die RP-ID gebundenen Elemente der Liste aus. Alle gewählten PKCSs werden im Folgenden als PKCS-Kandidaten bezeichnet. Nachdem mögliche Erweiterungen und weitere optionale Parameter verarbeitet wurden, muss im nächsten Schritt eine Nutzeraktion stattfinden. Abhängig von den gesetzten Flags besteht diese aus der Verifikation des Nutzers oder einem Test auf Anwesenheit. Alle weiteren Schritte dürfen erst nach dieser Aktion ausgeführt werden, da der Authentifikator erst nach dieser die Existenz einer PKCS preisgeben darf [8].
Falls die „allowCredentials“-Liste übergeben wurde und mindestens ein PKCS-Kandidat existiert, wählt der Authentifikator nach erfolgreicher Nutzeraktion eine passende PKCS aus den Kandidaten aus [8]. Die Liste der „allowCredentials“ ist dabei absteigend nach der Präferenz der Relying Party sortiert. Um den privaten Schlüsselanteil der PKCS zu ermitteln, entschlüsselt der Authentifikator die vom Server empfangene ID. Wurden keine „allowCre-

dentials“ übermittelt und mehrere resident PKCSs können verwendet werden, so wird der Nutzer aufgefordert, anhand der jeweils zugehörigen Nutzer-ID und möglicherweise einem Anzeigenamen die gewünschte PKCS auszuwählen [8]. Auf den privaten Schlüsselanteil der resident PKCS kann direkt zugegriffen werden. Ein globaler oder der PKCS zugeordneter Signaturzähler wird um einen positiven Wert inkrementiert, falls der Authentifikator über einen solchen verfügt. Als nächstes generiert der Authentifikator die „authenticatorData“. Dies ist ein Bytearray, welches einen Hash der RP-ID beinhaltet, auf welche die PKCS beschränkt ist, und Flags, welche angeben, ob die Nutzerverifikation beziehungsweise der Test auf Nutzerpräsenz erfolgreich war. Des Weiteren sind der Signaturzähler und mögliche Antworten auf Erweiterungen enthalten. Das „attestedCredentialData“-Feld wird an dieser Stelle nicht verwendet, sondern ist lediglich für die Registrierung von Bedeutung [4].
Im letzten Schritt wird anhand des privaten Schlüssels der ausgewählten PKCS eine Signatur S erzeugt. Als Input für die Operation dient die Konkatenation der „authenticatorData“ A und der Hash der JSON-serialisierten „clientData“ H. Die Signatur berechnet sich also zu $S = signature(A \mid\mid H)$ [4].

Antwort des Authentifikators

In Schritt 4 liefert der Authentifikator die vorher generierten „authenticatorData“ und die Signatur zurück an den Browser. Falls keine „allowCredentials“ übergeben wurden oder diese mehr als ein Element beinhalteten, liefert der Authentifikator des Weiteren die ID der verwendeten PKCS zurück. Wurde eine resident PKCS verwendet, muss an dieser Stelle ebenfalls die ID des Nutzers, welchem die PKCS zugeordnet ist, zurückgeliefert werden [8]. Weitere Rückgabewerte sind möglich und werden in Abschnitt 5.2 von [8] beschrieben.
Über den Rückgabewert des `get()`-Aufrufs leitet der Browser wiederum die Antwort des Authentifikators zusammen mit den in Schritt 2 generierten „clientData“ in JSON-Repräsentation und Antworten auf eventuelle Clienterweiterungen an die RP-Applikation weiter. Schritt 5 besteht aus der Übertragung aller empfangenen Parameter an den Server durch die Applikation [4, 34].
Im letzten Schritt (6) verifiziert der RP-Server die Antwort des Authentifikators. Dafür testet er zunächst, ob die ID der verwendeten PKCS in den übermittelten „allowCredentials“ enthalten war und dem Nutzer, welcher authentifiziert werden soll, zugeordnet ist. Die Nutzer-ID muss, falls in der Antwort vorhanden, eben jenen Nutzer beschreiben. Falls dieser noch unbekannt ist, wird die ID zu seiner Identifizierung verwendet. Anhand der PKCS-ID kann der Server den öffentlichen Schlüsselanteil der PKCS, welchen er bei der Registrierung des Nutzers gespeichert hat, abrufen. Im nächsten Teilschritt überprüft er, dass die Felder der „clientData“ den erwarteten Werten entsprechen, also der Typ auf „webauthn.get“ gesetzt ist, die Challenge der ursprünglich gestellten Challenge entspricht und die Origin gleich der der Relying Party ist. Anschließend werden die Werte der „authenticatorData“ überprüft. Der RP-ID-Hash muss dem Hash der übertragenden RP-ID entsprechen, und das Flag, welches Nutzerpräsenz anzeigt, muss gesetzt sein.

Wurde Nutzerverifikation gefordert, so überprüft der Server, dass das entsprechende Flag ebenfalls gesetzt ist. Anschließend werden die einzelnen Antworten auf Erweiterungen auf ihre Korrektheit überprüft. Anhand des öffentlichen Schlüsselanteils der für die Assertion verwendeten PKCS kann der RP-Server kontrollieren, ob die Signatur über die empfangenen Daten korrekt ist. Sollte der Signaturzähler der Antwort *new* oder der mit den PKCS-Informationen auf dem RP-Server gespeicherte Wert *old* ungleich 0 sein, zeigt dies an, dass der Authentifikator einen Signaturzähler unterstützt oder in der Vergangenheit unterstützt hat. Die Werte müssen in diesem Fall miteinander verglichen werden. Ist $new > old$, so wird *old* mit dem Wert von *new* überschrieben. Ist $new < old$, ist das ein Hinweis darauf, dass separate Kopien des privaten Schlüssels der PKCS existieren, also der Authentifikator möglicherweise dupliziert wurde. Diese Information geht in die Risikoberechnung einer Relying Party ein. Ob die Authentifizierung deshalb fehlschlägt oder der neue Wert übernommen wird, ist der Relying Party überlassen. Wurden alle vorangegangenen Schritte erfolgreich beendet, ist die Authentifizierung abgeschlossen.

4.4 Anwendungsfälle

Für die FIDO2-Authentifizierung sind verschiedene Anwendungsfälle möglich. Mit Hilfe eines Nutzernamens kann sie als zweiter oder alleiniger Faktor verwendet werden [54]. Nach der Eingabe des Nutzernamens und, falls gefordert, des Passwortes, kann der Server die öffentlichen Informationen über die auf den Nutzer registrierten PKCSs abrufen. Ihre IDs übermittelt er in den „allowCredentials“ an den Authentifikator. Dieser entschlüsselt die darin enthaltenen Informationen, bevor er eine Signatur generiert [55].
Einen Nutzer ohne Nutzernamen sondern lediglich mit Hilfe seiner ID zu authentifizieren ist unter Verwendung von resident PKCS möglich. Bei der Generierung einer Assertion mit einer resident PKCS, wird die ID des zugeordneten Nutzers zurückgeliefert. Dadurch ist es für den RP-Server möglich, den Nutzer anhand der Antwort in einem Schritt zu identifizieren und zu authentifizieren. Ein Nutzername ist dafür nicht notwendig. Die Korrelierbarkeit zwischen Nutzerkonten unterschiedlicher Relying Parties wird somit weiter reduziert. Ein Authentifikator, welcher resident PKCS speichert, kann diese ausschließlich anhand der RP-ID des Servers auszuwählen. Eine Liste von nutzerspezifischen „allowCredentials“ muss für die Authentifizierung nicht übermittelt werden. Dieser Anwendungsfall kann nur mit FIDO2-Authentifikatoren, nicht jedoch mit älteren U2F-Geräten, ausgeführt werden, da diese über keine Möglichkeit verfügen eine Nutzer-ID zu speichern und ohne „allowCredentials“-Eintrag nicht der nötige Key Handle generiert werden kann [4, 8].
Durch die lokale Verifikation gegenüber dem Authentifikator können der Authentifizierung weitere Faktoren hinzugefügt werden [4].

5 Die FIDO2-Erweiterung für TLS 1.3

Das Kernstück dieser Arbeit befasst sich mit der Erweiterung des TLS-1.3-Handshakes um clientseitige FIDO2-Authentifizierung. Im Folgenden wird diese zunächst noch abstrakte Erweiterung als FIDO2-Erweiterung und der TLS-Handshake unter Verwendung dieser als TFE[2]-Handshake bezeichnet.
Seinen Namen verdankt die FIDO2-Erweiterung dem Mechanismus, der seine Verwendung erlaubt: Den TLS-Erweiterungen. Über Erweiterungen kann in TLS die Verwendung bestimmter Funktionen zwischen Client und Server ausgehandelt werden [42] – in diesem Fall die Verwendung von clientseitiger FIDO2-Authentifizierung. Die Funktionalität der FIDO2-Erweiterung wird dabei jedoch nicht ausschließlich von Erweiterungen übernommen, sondern auch neue Nachrichten- und Fehlertypen werden definiert. Der Oberbegriff der FIDO2-Erweiterung, welche neue Erweiterungs-, Nachrichten- und Fehlertypen beinhaltet, ist also differenziert zu sehen zu einem konkreten Erweiterungstypen, welcher eine bestimmte Erweiterung zu einer Handshake-Nachricht beschreibt. Um Verwechslungen auszuschließen, beinhalten die Namen der nachfolgend definierten Erweiterungstypen daher den Nachrichtentypen der zugehörigen Nachricht. Eine konkrete Erweiterungen zur Ausübung von FIDO2-Funktionalität in der ClientHello-Nachricht wird so zum Beispiel FIDO2ClientHello-Erweiterung genannt.
Die FIDO2-Erweiterung fügt TLS eine weitere Methode der clientseitigen Authentifizierung hinzu. Sie soll die passwortlose Verwendung von FIDO2 ohne Nutzernamen zulassen. Auch der Einsatz mit Nutzernamen soll zur Unterstützung von U2F-Authentifikatoren möglich sein. Weitere Faktoren können der Authentifizierung auf verschiedene Weisen hinzugefügt werden. Es kann eine lokale Verifikation gegenüber dem Authentifikator stattfinden oder weitere Mechanismen, wie beispielsweise eine Passwortabfrage, auf Anwendungsschichtebene implementiert werden. Im ersten Fall finden optionale Parameter des FIDO2-Nachrichtenaustausches Verwendung. Der zweite Fall läuft von der FIDO2-Erweiterung getrennt ab und ist somit für diese irrelevant. Die Registrierung von Nutzern wird nicht berücksichtigt. Sie kann über eine Webschnittstelle stattfinden oder sich mit Modifikationen den nachfolgend beschriebenen Mechanismen bedienen. Für die beschriebene Authentifizierung wird eine vorherige Registrierung also bereits angenommen. Möglich soll die Verwendung der FIDO2-Erweiterung ab Version 1.3 von TLS sein.
Bevor die FIDO2-Erweiterung im Detail beschrieben wird, sollen zunächst in 5.1 die theoretischen Aspekte der Integration von FIDO2-Authentifizierung in den TLS-Handshake betrachtet werden. Erwartete Vor- und Nachteile jeder theoretischen Erweiterung werden dargelegt und anhand dieser Anforderungen an eine konkrete Umsetzung formuliert. In 5.2 werden unterschiedliche Strategien für die Realisierung der FIDO2-Erweiterung skizziert. Anschließend wird in 5.3 die gewählte Strategie und die für diese notwendigen

[2]TLS 1.3 with FIDO2 Extension

Änderungen am Handshake im Detail erläutert, bevor abschließend die Proof-Of-Concept-Implementation in 5.4 vorgestellt wird und praktische Aspekte der Erweiterung anhand ihrer diskutiert werden.

5.1 Theoretische Erwägungen

5.1.1 Vorteile

Kryptographiebasierte Verfahren wie FIDO2 bieten, wie bereits beschrieben, zahlreiche Vorteile gegenüber klassischer Passwortauthentifizierung. Eine TLS-Erweiterung zur Umsetzung von FIDO2-Authentifizierung erlaubt es prinzipiell jeder Anwendung, die eine TLS-Implementation verwendet, welche die Erweiterung unterstützt, FIDO2 zur Nutzerauthentifizierung anzubieten. Dadurch könnte FIDO2 häufiger zum Einsatz kommen. Im Idealfall könnten Nutzer komplett auf Passwörter verzichten und vollständig auf die passwortlose Authentifizierung umsteigen. Dies würde die Nutzung auch für Anwender attraktiver machen, welche für all ihre Accounts bei Onlinediensten das gleiche einheitliche Verfahren nutzen möchten. Die Vorteile von FIDO2 könnten somit flächendeckend zum Tragen kommen. Dabei könnte sogar der ursprüngliche Wirkungsbereich von FIDO2 überstiegen werden. Als Teil des TLS-Protokolls wäre FIDO2 nicht wie bislang auf die Verwendung in HTTPS-Verbindungen beschränkt, sondern könnte zum Beispiel auch für die Anmeldung bei der Verwendung von VPN über TLS verwendet werden.
Als Methode zur clientseitigen Authentifizierung stellt die FIDO2-Erweiterung eine Alternative zu der Verwendung der in TLS bereits vorhandenen Clientzertifikate dar. Nutzer über TLS unterhalb der Anwendungsschicht zu authentifizieren, widerspricht also nicht gängiger Praxis. Tatsächlich entspricht es sogar einem der drei Hauptziele von TLS, nämlich der Endpunkt-Authentifizierung. Aus der Authentifizierung während des TLS-Verbindungsaufbaus ergibt sich des Weiteren für die Anwendungsschicht der Vorteil, dass alle Sicherheitsparameter der Verbindung mit Ende des Handshakes feststehen – inklusive des Authentifizierungsstatus des Nutzers.
Anwendungsschichtprogrammiererinnen und -programmierern wird die Integration von FIDO2 in ihre Anwendung erleichtert, sollten sie TLS als Transportschichtprotokoll verwenden. Anstatt alle Bestandteile von FIDO2 korrekt einbinden und umsetzen zu müssen, können bei der Konfiguration der zugrundeliegenden TLS-Verbindung wenige zusätzliche Parameter angegeben werden, um die neue Authentifizierungsmethode anzubieten. Ein Beispiel für eine solche Konfiguration wird im weiteren Verlauf der Arbeit geliefert. Da auf diese Weise nicht die Entwicklerinnen und Entwickler jedes Onlinedienstes für die Unterstützung von FIDO2 auf Anwendungsschichtebene Funktionalität selbst implementieren müssen, sondern lediglich wenige Expertinnen und Experten die Integration der Erweiterung in TLS-Implementationen vornehmen müssen, wird die Anfälligkeit für Fehler in der Integration von FIDO2 verringert.

5.1.2 Nachteile

Änderungen an einem Protokoll gehen immer mit dem Risiko einher, neue Fehlerquellen einzuführen und somit Sicherheitslücken zu schaffen und neue Angriffsmöglichkeiten zu eröffnen. Dies gilt insbesondere, wenn aufwändige Mechanismen wie die FIDO2-Authentifizierung eingebunden werden sollen, welche dem Protokoll weitere Komplexität hinzufügen. Bei einem Protokoll wie TLS, welches zu den weit verbreitetsten Sicherheitsprotokollen im Internet zählt [51] und jährlich Onlinegeschäfte in mehrstelliger Milliardenhöhe absichert [32], können solche Sicherheitslücken schwerwiegende Folgen nach sich ziehen.
Bei einer weiten Verbreitung der FIDO2-Erweiterung muss FIDO2-Funktionalität nicht mehr von jedem Anwendungsschichtprogramm eingebunden werden, sondern lediglich für TLS-Implementationen, welche die Erweiterung unterstützen. Neben allen genannten Vorteilen birgt dies jedoch auch Risiken. Werden Fehler in verbreiteten Implementationen von TLS gemacht, so können diese zu weitreichenden Problemen führen und die Sicherheit der Authentifizierung vieler Dienste gleichzeitig beeinträchtigen.
Dem TLS-Handshake wird auch bei einem effizienten Entwurf der FIDO2-Erweiterung Latenz hinzugefügt, da sowohl auf Seite des Clients als auch des Servers zusätzliche kryptographische Operationen nötig sind, auf Daten zugegriffen werden muss und zusätzliche Daten übertragen werden müssen. Dies stellt ein Risiko für die Nutzerakzeptanz dar, da diese für Privatsphäre- und Sicherheitsanwendungen wie in [9] am Beispiel des Tor-Browsers gezeigt wird, stark von der Latenz abhängt. Besonderes auf mobilen Geräten macht sich die zusätzliche Latenz bemerkbar [35].
Auch die Umgewöhnung der Nutzer an eine neue Authentifizierungsmethode könnte ein Problem für die Akzeptanz darstellen. Besonders bezüglich der Eingabe von persönlichen Informationen ist eine Umstellung nötig. Die Authentifizierung über die FIDO2-Erweiterung erfolgt während des TLS-Handshakes, also bevor Server und Client Anwendungsdaten ausgetauscht haben. Im Falle einer Webseite bedeutet das beispielsweise, dass der Nutzername zusammen mit der Adresse des Servers an den Browser übergeben und die Authentifizierung bestätigt wird, bevor wie gewohnt die graphische Oberfläche der Webseite mit der Aufforderung zur Eingabe von Nutzerinformationen angezeigt werden kann.

5.1.3 Anforderungen

Aus den genannten Vor- und Nachteilen ergeben sich verschiedene Anforderungen an die FIDO2-Erweiterung für TLS 1.3. Die wichtigste ist die sichere Umsetzung der FIDO2-Authentifizierung im TLS-Handshake. Das bedeutet, dass keine der Sicherheitseigenschaften von TLS oder von FIDO2 durch die Kombination der beiden eingeschränkt werden soll. Durch die wichtige Bedeutung des TLS-Protokolls, ist es insbesondere entscheidend, dass bestehende Mechanismen des Protokolls durch die Erweiterung nicht beeinflusst werden. Maßgeblich ist daher, dass die Generierung von Schlüsselmaterial, die Aushandlung von

Serverparametern, serverseitige Authentifizierung und die Integritätsprüfung unverändert erhalten bleiben. Die FIDO2-Erweiterung soll darüber hinaus keine neuen Möglichkeiten schaffen, das Verhalten von Personen im Internet nachzuverfolgen. Sensible Informationen wie Nutzernamen sollten daher nur verschlüsselt übertragen werden.
Um neue Fehlerquellen und damit einhergehende Sicherheitsprobleme zu vermeiden, soll die Komplexität des TLS-Handshakes durch die FIDO2-Erweiterung nicht oder nur geringfügig erhöht werden. Dafür empfiehlt es sich, alle neu definierten Mechanismen an bereits in TLS existierenden Vorgängen auszurichten. Das heißt in diesem Fall, dass die mit der FIDO2-Erweiterung eingeführte Methode zur clientseitigen Endpunktauthentifizierung sich an bereits in TLS vorhandenen Mechanismen zur Umsetzung dieser orientieren soll.
Um die Verwendung der FIDO2-Erweiterung möglichst simpel zu halten und so eine weite Verbreitung zu ermöglichen, soll die Schnittstelle von TLS zur Anwendungsschicht weiterhin klar und einfach strukturiert bleiben. Auch die Kompatibilität zu Implementationen, welche die Erweiterung nicht unterstützen, muss gewährleistet sein.
Da erwartet wird, dass die Nutzerakzeptanz auch von der Latenz abhängt, besteht ein nachgeordnetes Ziel darin, die neu eingeführt Latenz im TLS-Handshake zu minimieren. Die Latenz, welche durch Client- und Serveroperationen entsteht, kann durch bessere Hardware behoben werden, genau wie die Bandbreite durch bessere Leitungen erhöht werden kann [38]. Eine klare untere Grenze ist jedoch dem Propagation Delay gesetzt [38], also der Zeit, welche ein Bit benötigt, um über eine Leitung von Router A zu Router B übertragen zu werden [32]. Diese Übertragung kann niemals schneller als die Lichtgeschwindigkeit erfolgen. Um das Propagation Delay möglichst gering zu halten, ist es daher wichtig die Anzahl der Roud Trip Times (RTTs) einer Verbindung zu minimieren [38]. Für den TLS-Handshake bedeutet dies, dass die Anzahl der Übertragungsphasen möglichst gering gehalten werden soll.
Werden von der FIDO2-Erweiterung und deren Implementationen alle aufgeführten Anforderungen erfüllt, so ist es möglich, die Folgen vieler der vorher genannten Nachteile minimal zu halten.

5.2 Kandidaten für den TFE-Handshake

Für die Handshake-Kandidaten wird, wie schon beim gewöhnlichen TLS-Handshake, ein Verbindungsaufbau zwischen Server und Client vorgestellt. Der Server bietet wie zuvor einen Onlinedienst an und führt den serverseitigen Anteil des TLS-Handshakes aus. Darüber hinaus muss er in der Lage sein FIDO2-Operationen auszuführen, also insbesondere die Generierung einer zufälligen Challenge und das Überprüfen der Antwort des Authentifikators. Ob der Server diese Aufgaben selbst ausführt oder über Kommunikation zu einem separaten Relying-Party-Server ist an dieser Stelle nicht relevant. Als Client wird weiterhin eine Anwendung bezeichnet, welche den Onlinedienst eines Servers in Anspruch nimmt und den clientseitigen Anteil des TLS-Handshakes ausführt. Wie der Server muss der Client im Folgenden in der Lage sein FIDO2-Funktionalität auszuführen. Er muss also

insbesondere nicht nur über einen Kommunikationskanal zu dem Server, sondern auch zu einem Authentifikator verfügen. Für die FIDO2-Authentifizierung wie in Abbildung 4.3 auf Seite 18 dargestellt, ersetzt er die Funktion der JavaScript-Applikation und des Browsers.
Eine *Angreiferin* oder ein *Angreifer* wird definiert als eine Person, welche das Netzwerk zwischen Server und Client kontrolliert, also Datenpakete unbemerkt lesen, löschen, verändern oder injizieren kann, wie in Abschnitt 3 von [43] beschrieben. Sie hat jedoch keinen Zugriff auf die Kommunikation zwischen dem Client und dem Authentifikator. Da der Informationsaustausch zwischen diesen zwei Parteien im TFE-Handshake unverändert gegenüber der gewöhnlichen FIDO2-Authentifizierung über einen bereits etablierten Kanal stattfindet, werden Angriffe auf diesen nicht berücksichtigt.
Um den TFE-Handshake zu ermöglichen, müssen Informationen, welche während der FIDO2-Authentifizierung für gewöhnlich zwischen RP-Server und JavaScript-Applikation ausgetauscht werden, innerhalb des TLS-Handshakes zwischen Server und Client übertragen werden. Das bedeutet, dass alle Informationen, welche in den Schritten 0, 1 und 5 der FIDO2-Authentifizierung nach Abbildung 4.3 übermittelt werden, in TLS-Handshake-Nachrichten verpackt werden müssen. Um die übertragenen Informationen zu charakterisieren, werden die zunächst abstrakten Objekte Indication, Username, Challenge und Response eingeführt. Die *Indication* beinhaltet keine Informationen, sondern zeigt dem Empfänger lediglich die Verwendung der FIDO2-Erweiterung an. Das Objekt *Username* enthält den Nutzernamen der Person, welche sich über die Erweiterung authentifizieren möchte. Die Übertragung des Indication- oder Username-Objekts entspricht Schritt 0 der Authentifizierung. Die *Challenge* umfasst alle in Schritt 1 übermittelten Informationen, welche der Client benötigt, um mit Hilfe des Authentifikators eine Assertion zu generieren. Die Antwort des Authentifikators mit allen notwendigen Parametern ist in der *Response* enthalten. Die Übertragung dieses Objektes entspricht Authentifizierungsschritt 5. Die genaue Struktur der Objekte und ob sie in einer Erweiterung oder einer separaten Nachricht übermittelt werden, bleibt an dieser Stelle zunächst unspezifiziert. Stattdessen soll lediglich der Ablauf der unterschiedlichen Handshake-Kandidaten gezeigt werden.
Jeder Kandidat beinhaltet zwei verschiedenen Modi zur Ausführung. Wird FIDO2 mit Hilfe von resident PKCS ohne Nutzernamen verwendet, so muss in Schritt 0 lediglich das Indication-Objekt übertragen werden. Dieser Modus wird als *FIDO2 with ID* (FI)-Modus bezeichnet. Wird ein Nutzername verwendet, zum Beispiel um ein U2F-Token nutzen zu können, so wird in Schritt 0 das Username-Objekt versandt. Diese Verwendungsart heißt *FIDO2 with Name* (FN)-Modus. Der wichtigste Unterschied für die beiden Handshake-Modi besteht darin, dass das Indication-Objekt unverschlüsselt übertragen werden kann, da es keine sensiblen Informationen enthält, wohingegen das Username-Objekt stets verschlüssel werden sollte, um Nutzerverfolgung auszuschließen. Die zusätzliche Verwendung von lokaler Verifikation gegenüber dem Authentifikator kann in beiden Modi stattfinden, ohne den Ablauf des Handshakes zu verändern.

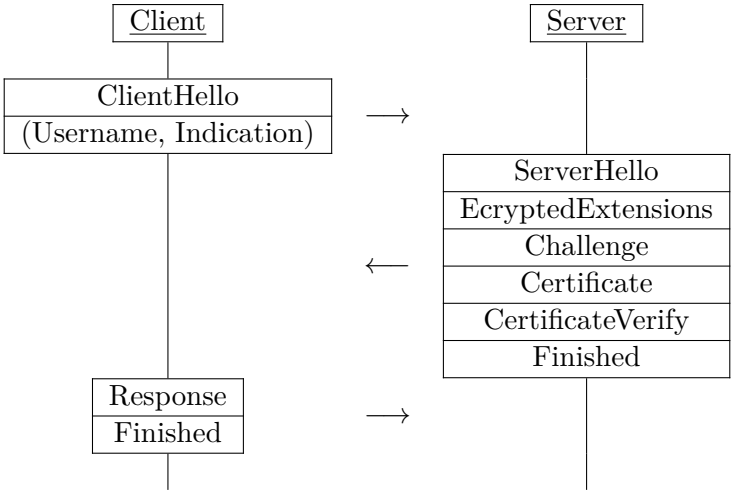

Abbildung 5.1: Einfacher Handshake
Quelle: Eigene Abbildung

Zwei durch Komma getrennte Objekte in runden Klammern zeigt in den folgenden Abbildungen an, dass genau eins der beiden Objekte übertragen werden muss. Das erste im Fall des FN- und das zweite im Fall des FI-Modus. Objekte in geschweiften Klammern müssen nur im FN-Modus übermittelt werden.

5.2.1 Einfacher Handshake mit statisch verschlüsseltem Nutzernamen

Beim ersten Kandidaten für den TFE-Handshake wird im FN-Modus der Nutzername mit dem öffentlichen Langzeitschlüssel des Servers verschlüsselt und zusammen mit dem ClientHello an diesen übertragen. Im FI-Modus wird stattdessen das unverschlüsselte Indication-Objekt verschickt. Der Server übermittelt die Challenge zusammen mit seiner ServerHello-Nachricht. Die Response des Clients wird vor dessen Finished-Nachricht versendet. Der Handshake wird in Abbildung 5.1 graphisch dargestellt.
Dieser Kandidat benötigt keine zusätzlichen RTTs gegenüber dem gewöhnlichen TLS-Handshake. Er fügt dem Verbindungsaufbau nur wenig Komplexität hinzu, da die bestehenden Nachrichtenphasen zwischen Client und Server genutzt werden, um zusätzliche Informationen zu übermitteln. Ein klarer Nachteil dieser Handshakevariante ergibt sich jedoch aus der Verschlüsselung mit dem statischen Serverschlüssel. Der Nutzername wird auf diese Weise nicht forward secure übertragen. Zeichnet ein potentieller Angreifer also alle eingehenden und ausgehenden Nachrichten des Servers auf (full take) und gelingt es ihm im Anschluss, den privaten Schlüssel zu dem öffentlichen Schlüssel des Servers zu berechnen, so ist er in der Lage, die Anmeldungen der verschiedenen Nutzer nachzuvollziehen.

5.2.2 Einfacher Handshake mit dynamischem Nutzernamen

Die Einschränkungen durch die statische Verschlüsselung des Nutzernamens im FN-Modus behebt die zweite TFE-Handshakevarianate. Die Nachrichtenabfolge entspricht der des vorangegangenen Handshakes, wie in Abbildung 5.1 dargestellt. Anstatt den

Nutzernamen im ClientHello statisch verschlüsselt zu übertragen, wird jedoch ein zufällig erzeugter dynamischer Nutzername unverschlüsselt übermittelt. Bei der Registrierung wird dieser initial festgelegt und bei jeder weiteren erfolgreichen Anmeldung neu vergeben. Bei der k-ten Anmeldung wird dem Client also der dynamische Nutzername für die $k+1$-te Anmeldung mitgeteilt. Da der neue dynamische Name vom Server stets zufällig gewählt werden muss, kann ein Angreifer vom $k+1$-ten Nutzernamen nicht auf den k-ten Nutzernamen zurückschließen und das Verhalten eines Nutzers somit nicht anhand des Namens nachverfolgen. Der neue Nutzername kann vom Server zusammen mit der Challenge übertragen werden. Er darf jedoch erst final gespeichert werden, nachdem der Client die Challenge korrekt beantwortet hat. Am FI-Modus entstehen keine Änderungen gegenüber dem vorherigen Kandidaten.
Während diese Handshakevariante das Problem der potentiellen Nutzerverfolgung löst, fügt es dem System auch weitere Komplexität hinzu. Der dynamische Nutzername muss vom Client gespeichert und zwischen verschiedenen Geräten synchronisiert werden.

5.2.3 Erweiterter Handshake

Das Problem, welches beide vorangegangenen Kandidaten zu lösen versuchen, besteht in der Übertragung sensibler Daten – in diesem Fall des Nutzernamens – bevor Client und Server die Chance hatten, ein ephemeres Geheimnis zu etablieren. Der erweiterte Handshake umgeht dieses Problem, indem er vertrauliche Daten erst überträgt, nachdem das Geheimnis erzeugt wurde. Dies ist der Fall, nachdem ClientHello und ServerHello ausgetauscht wurden [42].
Abbildung 5.2 zeigt den Ablauf des Handshakes. Der Client wartet mit der Übertragung des Nutzernamens im FN-Modus bis zum Empfang der ServerHello-Nachricht. Anschließend kann er den Namen forward secure verschlüsselt an den Server übermitteln. Um den Nachrichtenaustausch in beiden Modi analog zu gestalten, wird auch die Übertragung des Indication-Objektes im FI-Modus bis zu diesem Zeitpunkt verzögert. Der Server generiert die Challenge und sendet sie an den Client, welcher wiederum mit seiner Response antwortet. Die Finished-Nachrichten von Client und Server werden verzögert, bis sie ihre jeweils letzte Nachricht verschickt haben. Dem Handshake wird so insgesamt eine Nachrichtenaustauschphase hinzugefügt.
Der Vorteil des Kandidaten besteht darin, dass alle sensiblen Informationen durch das ephemere Geheimnis zwischen Client und Server geschützt sind. Es entsteht jedoch zusätzliche Latenz und Komplexität. Pro Handshake ist eine weitere RTT von Nöten und durch die zusätzliche Nachrichtenphase weicht die Struktur des Handshakes von der des gewöhnlichen TLS-1.3-Handshakes ab, wodurch mehr Zustände innerhalb des Protokolls erforderlich sind.

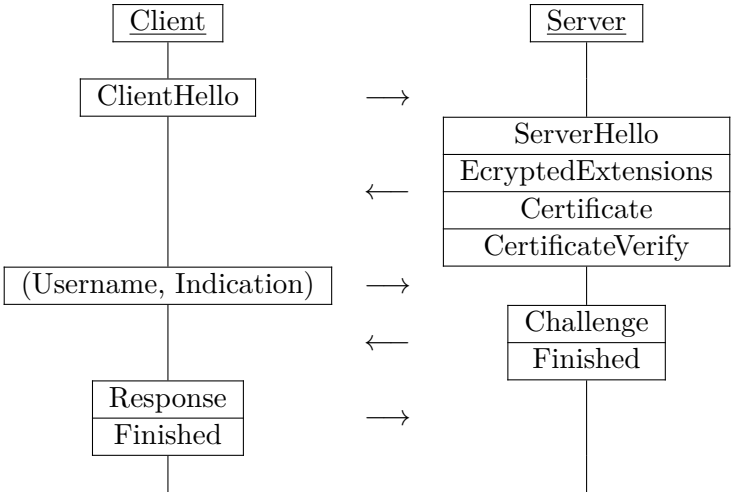

Abbildung 5.2: Erweiterter Handshake
Quelle: Eigene Abbildung

5.2.4 Doppelter Handshake

Die letzte Handshakevariante, welche vorgestellt werden soll, um FIDO2-Authentifizierung direkt im TLS-Handshake auszuführen, ist der doppelte Handshake. Er wird in Abbildung 5.3 dargestellt. Im FN-Modus wird nicht nur ein Handshake, sondern zwei ausgeführt. Der erste Handshake erfüllt den Zweck, den Nutzernamen sicher verschlüsselt an den Server zu übertragen und dort zusammen mit einem ephemeren Nutzernamen, welcher Client und Server bekannt ist, zu speichern. Erst im zweiten Handshake findet die FIDO2-Authentifizierung statt. Der Client nutzt dabei den ephemeren Nutzernamen, um sich gegenüber dem Server zu identifizieren, welcher anhand dessen den Nutzernamen ermitteln kann. Im FI-Modus wird lediglich der zweite Handshake ausgeführt. Für den doppelten Handshake müssen zusätzliche Objekte definiert werden. Der *NameRequest* fordert einen Client auf, den Nutzernamen und einen flüchtigen Nutzernamen zu übertragen. In dem „*Uname + EphUname*“-Objekt sind genau diese Informationen enthalten. Die Indication muss für diesen Kandidaten den zu verwendenden Modus anzeigen.
Im ersten Handshake des FN-Modus verschickt der Server, nachdem der Client die Nutzung der FIDO2-Erweiterung angezeigt hat, zusammen mit der ServerHello-Nachricht einen NameRequest an den Client, um dessen Identität in Erfahrung zu bringen. In seiner Antwort überträgt dieser seinen Nutzernamen zusammen mit einem ephemeren Nutzernamen verschlüsselt an den Server. Dieser speichert das Tupel aus temporärem und dauerhaftem Namen für eine begrenzte Zeit ab. Die Verbindung wird anschließend beendet.
Im zweiten Handshake überträgt der Client zusammen mit dem ClientHello im FN-Modus seinen ephemeren Nutzernamen. Diesen kann der Server verwenden, um den dazugehörigen Nutzernamen abzurufen. Nun kann er für diesen Nutzer eine Challenge übermitteln. Im FI-Modus beginnt der Client direkt mit dem zweiten Handshake und deutet durch die Indication lediglich die Nutzung der Erweiterung an. Der Server generiert die Challenge in diesem Fall nutzerunabhängig. Der Client antwortet in beiden Modi zusammen mit seiner

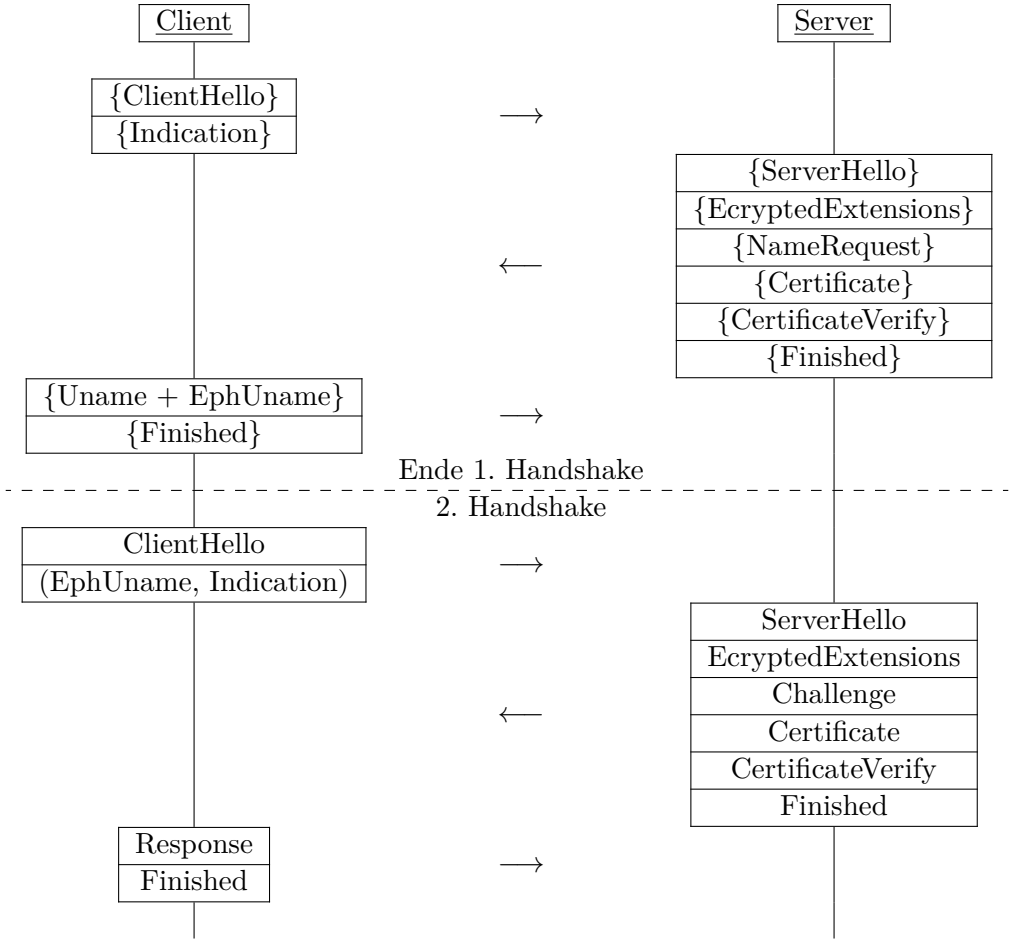

Abbildung 5.3: Doppelter Handshake
Quelle: Eigene Abbildung

Finished-Nachricht in der Response.
Der doppelte Handshake überträgt alle sensiblen Informationen forward secure verschlüsselt und benötigt keine Synchronisation zwischen unterschiedlichen Geräten. Jeder einzelne Handshake enthält die gleichen Übertragungsphasen wie der gewöhnliche TLS-Handshake. Offensichtlich wird dem Verbindungsaufbau jedoch zusätzliche Latenz hinzugefügt, da im FN-Modus zwei vollständige TLS-Handshakes ausgeführt werden.

5.2.5 Post-Handshake Authentifizierung

Eine Alternative zu allen bereits vorgestellten Handshake-Varianten stellt die Post-Handshake Authentifizierung dar. Bei dieser Methode wird die Nutzerauthentifizierung nicht im Handshake selbst ausgeführt. Stattdessen wird in diesem lediglich ausgehandelt, eine clientseitige Authentifizierung über FIDO2 nach dem Handshake auszuführen. Dafür muss während des Handshakes lediglich das *UsePostAuth*-Objekt zwischen Client und Server ausgetauscht werden, um die Verwendung des Verfahrens gegenseitig zu signalisieren. Die eigentliche Authentifizierung findet zu einem beliebigen Zeitpunkt nach dem Handshake statt. Abbildung 5.4 zeigt den Ablauf des Handshakes und der Authentifizierung. Durch

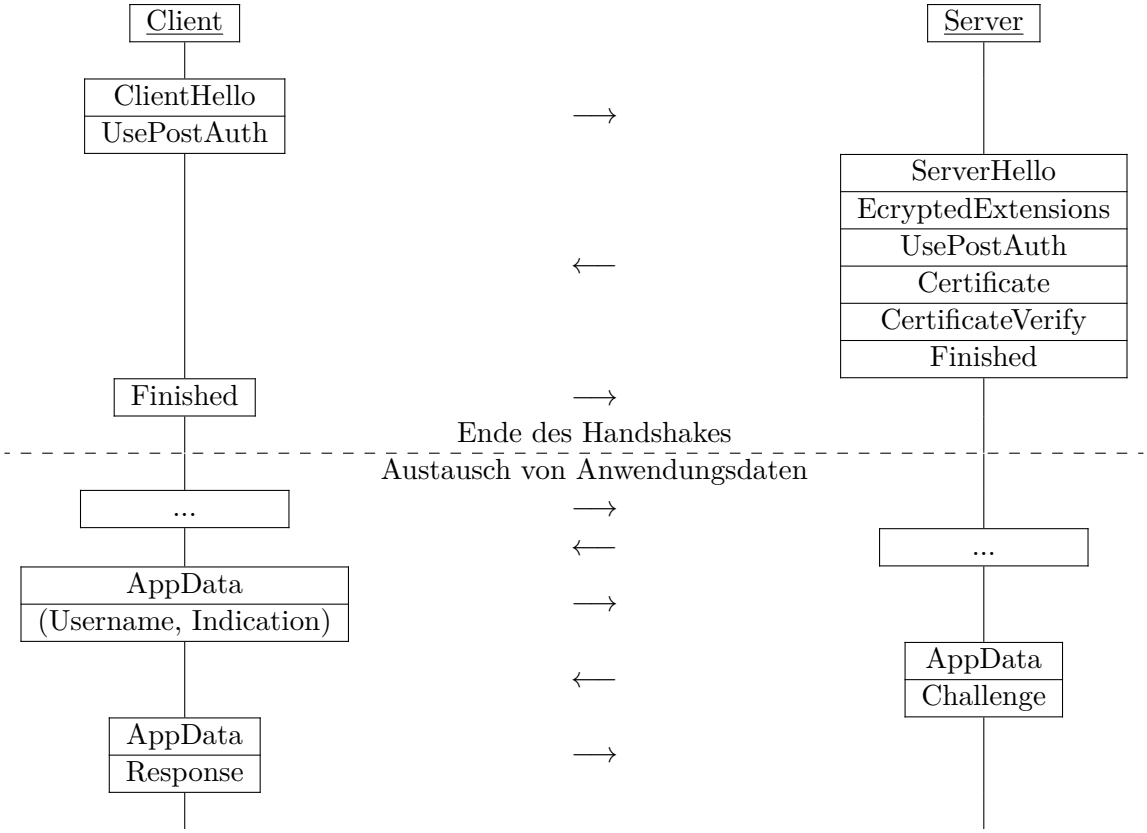

Abbildung 5.4: Post-Handshake Authentifizierung
Quelle: Eigene Abbildung

die mit drei Punkten gefüllten Felder wird angedeutet, dass beliebige Nachrichten zwischen Server und Client ausgetauscht werden.
Die Post-Handshake Authentifizierung erhöht die Komplexität des TLS-Handshakes kaum, da lediglich die Einigung auf eine spätere Authentifizierung zwischen Server und Client erfolgt. Dem Nachrichtenaustausch wird keine zusätzliche RTT und somit nur wenig Latenz hinzugefügt, da die für die Authentifizierung notwendigen Nachrichten zusammen mit Anwendungsdaten übertragen werden können. Ein Nachteil der Authentifizierungsmethode besteht jedoch darin, dass nicht alle Sicherheitsparameter der Verbindung mit Ende des Handshakes feststehen. Der Status der clientseitigen Authentifizierung kann sich zu einem beliebigen Zeitpunkt nach dem Handshake ändern.

5.3 TFE-Handshake: Der Gewinner

Die Wahl eines geeigneten Kandidaten soll durch den Vergleich der definierten Anforderungen mit den Eigenschaften der vorgestellten Kandidaten erfolgen. Die wichtigste Anforderung an den TFE-Handshake ist die sichere Umsetzung, also insbesondere auch die Forward Secrecy aller relevanten übertragenen Informationen. Dies schließt den einfachen Handshake mit statischer Nutzernamenverschlüsselung aus, da dieser bei einer Kompromittierung des Langzeit-Serverschlüssels die Entschlüsselung von Nutzernamen

erlaubt. Um Fehler im Entwurf und in Implementationen zu vermeiden, wurde gefordert, die Komplexität des Handshakes lediglich geringfügig zu erhöhen. Dies schließt den einfachen Handshake mit dynamischen Nutzernamen aus, welcher durch die Synchronisation des dynamischen Namens zwischen verschiedenen Geräten aufwendige Mechanismen und Zustandsänderungen verlangt. Ebenfalls der erweiterte Handshake wird ausgeschlossen, da er dem Handshake in beiden Modi eine weitere Nachrichtenphase hinzufügt und somit dessen grundlegenden Ablauf verändert. Sowohl der doppelte Handshake als auch die Post-Handshake Authentifizierung übertragen alle Informationen der FIDO2-Authentifizierung forward secure verschlüsselt und fügen dem einzelnen Handshake keine neuen Nachrichtenphasen hinzu. Zudem benötigen sie keine Synchronisation zwischen Geräten. Der doppelte Handshake erhöht die Latenz des Verbindungsaufbaus im FN-Modus deutlich mehr, wohingegen die Post-Handshake Authentifizierung eine Änderung der Sicherheitseigenschaften der Verbindung nach dem Ende des Handshakes zulässt. Da die Reduzierung der neu eingeführten Latenz in den Anforderungen als nachgeordnetes Ziel eingestuft wurde und die frühzeitige Feststellung aller Sicherheitsparameter von wichtiger Bedeutung ist, ist der Gewinner der **doppelte Handshake**. Der TFE-Handshake bedient sich der Struktur dieses Kandidaten und soll im Folgenden detaillierter vorgestellt werden. Im Abschnitt „Verbesserungen“ wird des Weiteren beschrieben, wie auch das nachgeordnete Ziel der geringen Latenzerhöhung eingehalten werden kann, ohne die restlichen Eigenschaften des Handshakes zu verändern.
Wie in den Anforderungen definiert, orientiert sich der TFE-Handshake an bereits bestehenden Mechanismen zur Clientauthentifizierung in TLS, nämlich den Clientzeritifikaten[3].

Terminologie
Den Schlüsselwörtern „MUSS“ (MUST), „DARF NICHT“ (MUST NOT), „SOLLTE“ (SHOULD), „SOLLTE NICHT“ (SHOULD NOT) und „KANN“ (MAY) kommt genau dann, wenn sie im Text in Großbuchstaben geschrieben vorkommen, eine besondere Bedeutung zu. Ihre Verwendung wird für die in Klammern angegebenen englischen Gegenstücke in [7, 33] definiert.

5.3.1 Ablauf im FI-Modus

Abbildung 5.5 zeigt den Ablauf des TFE-Handshakes im FI-Modus. Neu definierte Nachrichten- und Erweiterungstypen sind in blau markiert. Erstere werden dabei als eigenständige Box innerhalb einer Nachrichtenübertragungsphase, letztere getrennt durch eine gestrichelte Linie unterhalb der Nachricht, in der sie verschickt werden, dargestellt. Erweiterungen, welche in jedem TLS-1.3-Handshake übertragen werden, wie „supported_versions“, werden für eine bessere Übersichtlichkeit nicht dargestellt.

[3]Der Handshake unter Verwendung von Clientzertifikaten wird im Folgenden auch TLS-1.3 with Client Certificate Handshake (TCC-Handshake) genannt

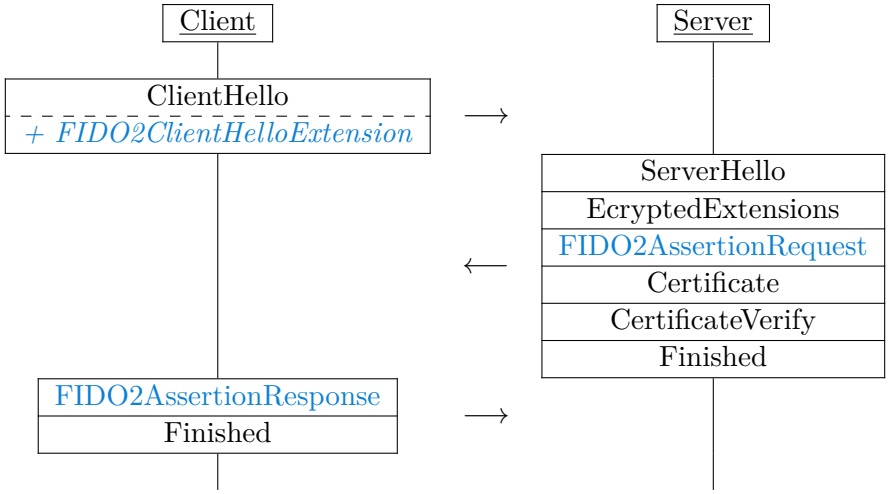

Abbildung 5.5: Der TFE-Handshake im FI-Modus
Quelle: Eigene Abbildung

Der Client initiiert den TFE-Handshake mit der ClientHello-Nachricht. Anhand der *FIDO2-ClientHelloExtension* signalisiert er dem Server gegenüber die Verwendung der FIDO2-Erweiterung. Dies entspricht Schritt 0 der Authentifizierung nach Abbildung 4.3. Das Format der ClientHello-Erweiterung wird in Anhang A.1 beschrieben. Sie beinhaltet den Modus, welchen der Client zu verwenden wünscht. Dieser MUSS auf FI gesetzt sein.
In seiner ServerHello-Nachricht und Erweiterungen übermittelt der Server alle notwendigen Parameter für die Generierung eines ephemeren Geheimnisses und leitet Schlüsselmaterial ab. Alle folgenden Nachrichten werden daher forward secure verschlüsselt. Sollte der Server die FIDO2-Erweiterung nicht unterstützen, so ignoriert er die FIDO2ClientHelloExtension und fährt mit dem gewöhnlichen TLS-Handshake fort. Sollte der Client die Erweiterung nicht übertragen haben, KANN der Server den Handshake mit einem „fido2_required“-Alert abbrechen. Unterstützen beide Endpunkte die Erweiterung, überträgt der Server in der gleichen Übertragungsphase wie das ServerHello eine *FIDO2AssertionRequest*-Nachricht. Dies entspricht Schritt 1 der Authentifizierung. Der Nachrichtentyp beinhaltet alle laut 5.5 in [4] möglichen Anfrageparameter des Servers für die FIDO2-Authentifizierung und wird in Anhang B.3 definiert. Insbesondere enthält er die Challenge des Servers. Da noch keine Authentisierung stattgefunden hat, wird die Anfrage nutzerunabhängig generiert. Sie beinhaltet somit keine Liste mit den IDs bereits auf den Nutzer registrierter PKCSs als „allowCredentials“. Wie im gewöhnlichen TLS-1.3-Handshake, überträgt der Server eine Certificate-, CertificateVerify- und Finished-Nachricht.
Die Schritte 2 bis 4 der FIDO2-Authentifizierung werden vom Client zusammen mit dem Authentifikator ausgeführt. Bevor dies geschieht, MUSS auch clientseitig das ephemere Schlüsselmaterial berechnet werden und die Identität des Servers sowie die Integrität des Handshakes anhand dessen Certificate-, CertificateVerify- und Finished-Nachricht überprüft werden. Dadurch wird sicher gestellt, dass nur der beabsichtigte Kommunikationspartner die im Folgenden übertragene Antwort des Clients entschlüsseln kann. Anschließend MUSS

der Client prüfen, ob die im FIDO2AssertionRequest übermittelte RP-ID der Domain des authentifizierten Servers oder einem registrierbaren Domain-Suffix entspricht. Ist dies nicht der Fall, MUSS er den Handshake mit einem „fido2_bad_request“-Alert abbrechen. Durch die Überprüfung der Identität des Servers und den Vergleich mit der übermittelten RP-ID werden Man-in-the-Middle-Angriffe auf die FIDO2-Authentifizierung unterbunden. Neben der Überprüfung der RP-ID MUSS der Client, wie in 5.1.4 von [4] beschrieben, des Weiteren die „clientData“ generieren, Erweiterungen verarbeiten und die notwendigen Parameter an die `authenticatorGetAssertion()`-Methode geeigneter Authentifikatoren weitergeben. Die Bestandteile der Origin für die „clientData“ können den Verbindungsinformationen entnommen werden. Um kompatibel zum WebAuthn-Standard zu bleiben, MUSS das Schema der Origin auf „https“ gesetzt werden. Schlägt die Generierung einer Assertion fehl, MUSS der Client den Verbindungsaufbau mit einem „internal_error“-Alert beenden. Hat der Nutzer den Vorgang abgebrochen, MUSS ein „user_canceled“-Alert verwendet werden. Bei erfolgreichem Abschluss der Operation werden die Rückgabewerte des Authentifikators gemeinsam mit den „clientData“ und Ergebnissen eventueller Erweiterungen in eine *FIDO2AssertionResponse*-Nachricht verpackt. Die ID des Nutzers, an welchen die verwendete PKCS gebunden ist, wird vom Authentifikator zurückgeliefert und MUSS als „user_handle“ in dieser Nachricht übermittelt werden. Die FIDO2AssertionResponse wird in der gleichen RTT wie die Finished-Nachricht an den Server gesendet, was Schritt 5 der Authentifizierung darstellt. Ihr Nachrichtenformat beinhaltet alle in 5.1 und 5.2.2 von [4] beschriebenen Antwortparameter und wird in Anhang B.4 definiert.
Der Server MUSS die Verifikation der Antwort nach FIDO2-Vorgaben in Schritt 6 der Authentifizierung ausführen. Den Nutzer identifiziert er anhand des „user_handle“-Eintrages der FIDO2AssertionResponse. Schlägt die Verifikation fehlt, MUSS der Server den Handshake mit einem „fido2_authentication_error“-Alert beenden. Nach der Risikoberechnung anhand der Überprüfung des Signaturzählers, des Authentifikatormodells und möglicherweise weiteren Parametern, KANN der Server den Handshake mit einem „fido2_authentication_error“-Alert abbrechen. Der Server KANN jedoch auch, sollte er den Nutzer nach der Risikoberechnung nicht als authentifiziert ansehen, den Handshake mit dem unauthentifizierten Nutzer fortsetzen. Folgen, welche sich daraus ergeben, werden in Abschnitt 5.3.4 besprochen. Ist die Verifikation der Antwort erfolgreich, so ist die FIDO2-Authentifizierung abgeschlossen. Nach der Prüfung der Finished-Nachricht ist der Handshake beendet und Anwendungsdaten können übertragen werden.

5.3.2 Ablauf im FN-Modus

Um die Unterstützung von U2F-Sicherheitstokens zu gewährleisten und FIDO2-Authentikatoren die Möglichkeit zu geben, Informationen auf den RP-Server auszulagern um mehr PKCSs erstellen zu können, wird der TFE-Handshake auch im FN-Modus angeboten. Dieser ermöglicht die Verwendung von nicht-resident PKCS, fordert diese jedoch nicht zwangsläufig. Abbildung 5.6 zeigt den Ablauf des TFE-Handshakes im FN-Modus. Im

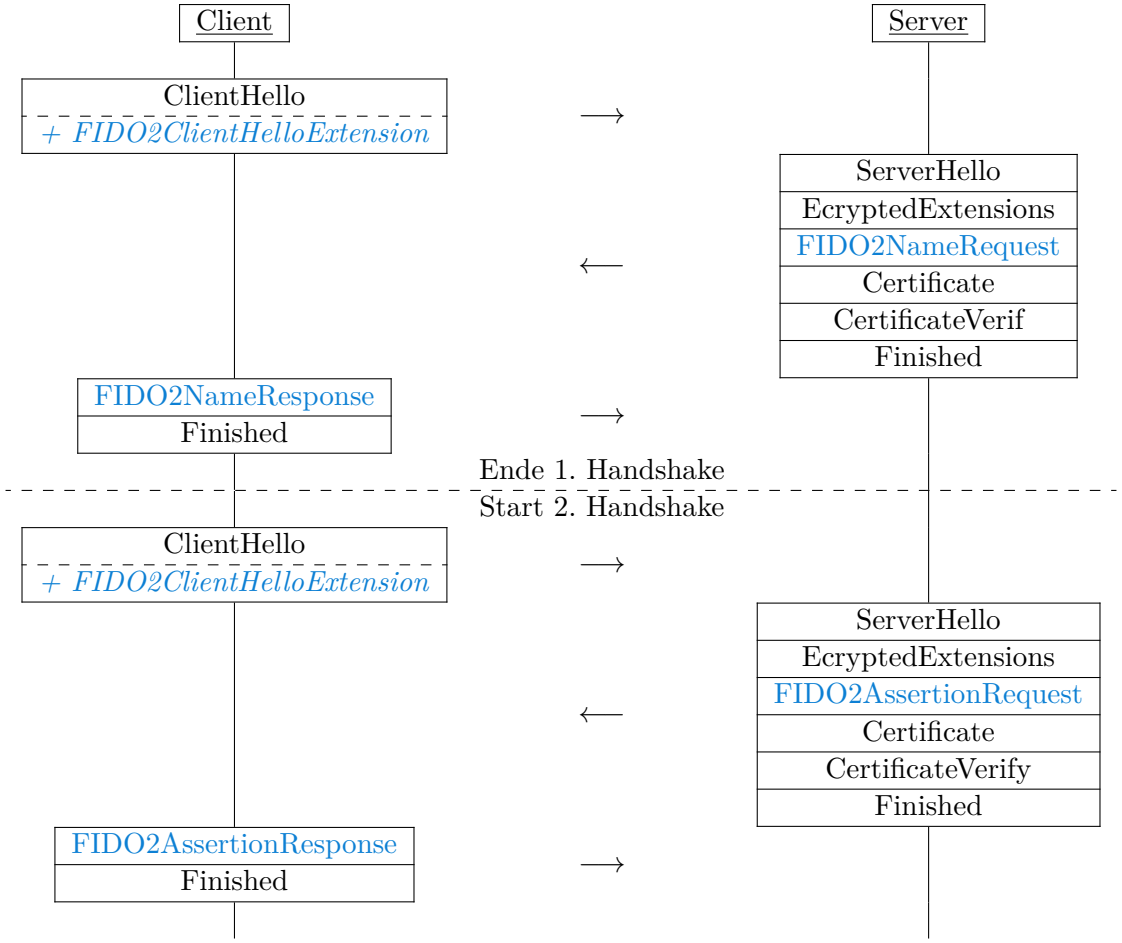

Abbildung 5.6: Der TFE-Handshake im FN-Modus
Quelle: Eigene Abbildung

Gegensatz zum FI-Modus muss, bevor der Server eine Anfrage stellen kann, zunächst der Nutzername übermittelt werden. Deshalb wird der volle doppelte Handshake ausgeführt.

Erster Handshake

Zu Beginn des Nachrichtenaustausches des ersten Handshakes signalisiert der Client in der *FIDO2ClientHelloExtension* die Verwendung von FIDO2-Authentifizierung. Der in der Erweiterung enthaltene Modus MUSS auf FN gesetzt sein. Server, welche die Erweiterung nicht unterstützen, ignorieren sie. Sollte der Server die FIDO2-Erweiterung unterstützen, überträgt er nach dem ServerHello in seiner ersten Nachrichtenphase eine *FIDO2NameRequest*-Nachricht. In dieser ist der Serveranteil S für die Generierung eines ephemeren Nutzernamen enthalten. Dieser besteht aus 32 vom Server zufällig gewählten Bytes. Der Client generiert ebenfalls 32 Zufallsbytes C und überträgt diese als Antwort in seiner zweiten Nachrichtenphase in einer *FIDO2NameResponse*-Nachricht. Neben dem Clientanteil beinhaltet diese außerdem den Nutzernamen[4] der zu authentifizierenden Person. Die FIDO2NameRequest- und FIDO2NameResponse-Nachrichten werden nach

[4]Im Folgenden wird der Nutzername auch als Langzeit-Nutzername bezeichnet um Verwechselungen mit dem ephemeren Nutzernamen auszuschließen

dem ServerHello verschickt und sind somit durch das ephemere Schlüsselmaterial geschützt, welches Server und Client zu diesem Zeitpunkt generiert haben.
Wurden die Anfrage- und Antwortnachrichten ausgetauscht, können Server und Client den ephemeren Nutzernamen E als den SHA-256-Hash der Konkatenation aus Server- und Clientanteil, also $E = SHA265(S \mid\mid C)$, berechnen. Im Gegensatz zu der zuvor beschriebenen Basisversion des doppelten Handshakes fließen also zufällig gewählte Bytes von Client und Server in die Berechnung von E mit ein. Weder der eine noch der andere kann den Wert somit alleine bestimmen. Dadurch wird es einem Angreifer deutlich erschwert, auf dem Server gespeicherte ephemere Nutzernamen gezielt zu überschreiben oder sie durch das Ausnutzen eines schwachen Zufallsgenerators von einem der Endpunkte zu erraten und das Anmeldeverhalten von Nutzern nachzuverfolgen.
Ist der vom Client übertragene Nutzername registriert, speichert der Server den ephemeren Nutzernamen, Langzeit-Nutzernamen[5] und die Lebenszeit des Eintrags. Diese DARF NICHT länger als 7 Tage sein. Dieser Zeitraum entspricht der Gültigkeitsdauer von PSK-Identitäten in TLS [42]. Sie SOLLTE vom Server für den beschriebenen Handshake auf maximal wenige Minuten begrenzt werden. Eine längere Lebensdauer ist erst für eine in 5.3.6 beschriebene Modifikation des Handshakes sinnvoll. Ist der Langzeit-Nutzername bei dem Onlinedienst des Servers nicht registriert, muss der Handshake dennoch weiterhin korrekt ablaufen. Andernfalls wäre es einem Angreifer möglich, für einen gegebenen Nutzer in Erfahrung zu bringen, ob dieser registriert ist oder nicht. Der Server DARF daher den Handshake NICHT aus diesem Grund abbrechen. Stattdessen MUSS er auch diesen Eintrag speichern, SOLLTE ihn jedoch als nicht registriert markieren.
Nach dem Ende des ersten Handshakes MUSS die Verbindung vom Client beendet werden. Auch die TCP-Verbindung SOLLTE terminiert werden. Ein Endpunkt ignoriert sämtliche Daten, welche nach Beendigung der TLS-Verbindung übertragen werden [42]. Der zweite Handshake MUSS daher über eine neue TCP-Verbindung erfolgen.

Zweiter Handshake

Der zweite Handshake entspricht in weiten Teilen dem TFE-Handshake im FI-Modus. Auch er beginnt mit dem Übertragen der *FIDO2ClientHelloExtension* durch den Client, was wiederum Schritt 0 des FIDO2-Authentifizierungsprozesses darstellt. In der Erweiterung ist der ephemere Nutzername enthalten, welchen Client und Server im vorherigen Handshake generiert haben. Der Modus MUSS auf FN gesetzt sein. Anhand des ephemeren Nutzernamens ermittelt der Server den dazu passenden Langzeit-Nutzernamen. Dabei MUSS er die Gültigkeitsdauer des Eintrags überprüfen. Ist der Eintrag nicht mehr gültig oder der ephemere Nutzername in der Datenbank nicht vorhanden, KANN der Server einen FIDO2NameRequest übermitteln. Es findet also ein Fallback auf den ersten Handshake statt und ein gültiger ephemerer Name für die nächste Verbindung kann generiert werden. Alternativ KANN er sich dazu entscheiden, den Verbindungsaufbau mit einem

[5] Anstatt des Langzeit-Nutzernamens KANN ein Server die ID eines Nutzers speichern.

„fido2_bad_request“-Alert zu beenden[6]. Der ephemere Nutzername besteht aus zufällig generierten Bytes und alle für seine Berechnung notwendigen Parameter wurden im ersten Handshake vertraulich ausgetauscht. Auch wenn er unverschlüsselt übertragen wird, kann daher aus ihm durch einen Angreifer nicht auf den Langzeit-Nutzernamen geschlossen werden. Nach einmaliger Verwendung MUSS er vom Server gelöscht werden. Alle weiteren für die FIDO2-Authentifizierung relevanten Nachrichten werden nach der ServerHello-Nachricht ausgetauscht, also nachdem Client und Server ephemeres Schlüsselmaterial generiert haben, und werden somit forward secure verschlüsselt.
Nachdem der Server den Nutzernamen, bzw. die dem Namen zugeordnete Nutzer-ID, ermittelt hat, kann er alle für die FIDO2-Authentifizierung notwendigen Parameter generieren. Neben dem Erstellen einer zufälligen Challenge MUSS er anhand des Nutzernamens auf die für diesen Nutzer registrierten PKCSs zugreifen und die „allowCredentials“ generieren. Alle weiteren Parameter werden wie im FI-Modus gesetzt und zusammen mit der Challenge und den „allowCredentials“ in die *FIDO2AssertionRequest*-Nachricht verpackt, welche in Schritt 1 der FIDO2-Authentifizierung an den Client übermittelt wird. Falls der Langzeit-Nutzername zu dem ephemeren Nutzernamen nicht zu einem registrierten Nutzer gehört, MUSS der Server dennoch einen FIDO2AssertionRequest übermitteln. Dies verhindert erneut, dass ein Angreifer für einen gegebenen Nutzer in Erfahrung bringen kann, ob er registriert ist. Der Server SOLLTE zudem Maßnahmen umsetzen, welche es erschweren, anhand der Anfrage-Parameter den Registrierungsstatus eines Nutzers zu erkennen. Mögliche Maßnahmen werden in Abschnitt 5.4 vorgestellt.
Die Überprüfung der vom Server übertragenen Parameter und die Generierung einer *FIDO2AssertionResponse*-Nachricht mit Hilfe des Authentifikators in den Authentifizierungsschritten 2 bis 5, findet wie zuvor für den FI-Modus beschrieben statt. Der einzige Unterschied besteht darin, dass das „user_handle“-Feld vom Client gesetzt werden KANN, allerdings nicht muss.
Auch die in Schritt 6 vollzogene Verifikation der Antwort durch den Server findet wie im FI-Modus statt. Dieser benötigt jedoch keine ID für die Identifizierung des Nutzers, da diese bereits über den ephemeren Nutzernamen erfolgt ist. Wie im FI-Modus MUSS der Server den Handshake mit einem „fido2_authentication_error“-Alert beenden, sollte die Verifikation fehlschlagen. Dies gilt im FN-Modus insbesondere auch, wenn die Authentifizierung für einen unregistrierten Nutzer veranlasst wurde. Der Client erfährt somit, dass mit dem verwendeten Paar aus Namen und PKCS keine Authentifizierung möglich ist. Ob der Name zu einem registrierten Nutzer gehört, bleibt jedoch verborgen. Nach der Risikoberechnung und der Überprüfung der Finished-Nachricht ist der TFE-FN-Handshake abgeschlossen. Alle Sicherheitsparameter der Verbindung, inklusive des clientseitigen Authentifizierungsstatus, stehen fest und Anwendungsdaten können ausgetauscht werden.

[6]Einem Angreifer ist es möglich in Erfahrung zu bringen, ob ein ephemerer Nutzername in der Datenbank des Servers existiert. Daraus sollte jedoch keine Informationsgewinnung über Nutzer möglich sein. Andernfalls könnte der Server unabhängig des Speicherstatus des ephemeren Nutzernamens eine FIDO2AssertionRequest-Nachricht übermitteln, um diese Information nicht preiszugeben.

5.3.3 Nachrichten-, Erweiterungs- und Alert-Typen

Alle für die FIDO2-Erweiterung benötigten Nachrichten-, Erweiterungs- und Alert-Typen werden im Anhang detailliert beschrieben. Insbesondere werden alle Bestandteile der Nachrichten und Erweiterungen erläutert.
Die für den TFE-Handshake definierten Nachrichten sowie deren Funktionen sind der Bedeutung der Nachrichten im Handshake mit Clientzertifikaten nachempfunden. Ähnlich zur CertificateRequest-Nachricht im TCC-Handshake fordern die FIDO2NameRequest- und FIDO2AssertionRequest-Nachrichten den Client zur Übermittlung von Informationen auf, welche für die Authentifizierung notwendig sind. Anstatt in einer Certificate- und CertificateVerify-Nachricht werden diese Informationen in den FIDO2NameResponse- und FIDO2AssertionResponse-Nachrichten übertragen. Da die FIDO2-Authentifizierung über den Erweiterungsmechanismus in den Standard eingebunden wird, ist das einzige Objekt, welches kein Gegenstück im TCC-Handshake hat, die FIDO2ClientHelloExtension, welche notwendig ist, um die Verwendung des Features zu signalisieren, den Modus sowie den ephemeren Nutzernamen zu übertragen.
Der Großteil der für den TFE-Handshake benötigten Informationen wird in neu definierten Nachrichten übermittelt und nur für einen kleinen Teil der Informationen wird eine Erweiterung verwendet. Einerseits erlaubt dies den Ablauf des TFE-Handshakes, wie beschrieben, möglichst analog zum TCC-Handshake zu gestalten. Andererseits löst dies Platzprobleme, welche bei der Verwendung von Erweiterungen entstünden. Da die maximale Länge der ID einer PKCS mit $2^{16} - 1$ Bytes [4] der maximalen Größe eines Erweiterungseintrages entspricht und nie mehrere Erweiterungen des gleichen Typen an eine Nachricht angehängt werden dürfen [42], wäre unter anderem die Übertragung einer Liste von PKCS-IDs als „allowCredentials“ in Erweiterungen nicht ohne Weiteres möglich.
Die neu definierten Nachrichten- und Erweiterungstypen sind grundsätzlich durch eine Menge von Einzelfeldern fester Reihenfolge aufgebaut. Diese Reihenfolge und die Codierung der enthaltenen Information muss also für das Parsing bekannt sein. Felder variabler Länge werden über ein Length-Value-Paar beschrieben. Das Vorhandensein optionaler Felder wird durch einzelne Bits im ersten Byte – dem Flag-Byte – signalisiert. Eine Alternative für diese Art der Nachrichtencodierung wäre die Modellierung der Nachrichten mittels ASN.1 und eine Codierung nach DER-Vorgaben. ASN.1 ist ein internationaler Standard, welcher verwendet wird, um Nachrichtenformate zu beschreiben, ohne die exakte Codierung selbst festlegen zu müssen, da diese über Codierungsregeln vorgeschrieben ist. Sowohl die Modellierung als auch die Codierung ist dabei unabhängig von der Rechnerarchitektur, dem Betriebssystem und der Programmiersprache [12].

5.3.4 Kommunikation zur Anwendungsschicht

Nach dem Ende eines Handshakes muss es aufseiten des Servers für die Anwendungsschicht möglich sein, den Authentifizierungsstatus und die Identität eines Nutzers in Erfahrung zu

bringen. Auch für diese Aufgabe orientiert sich die FIDO2-Erweiterung an Mechanismen, welche für Clientzertifikate genutzt werden. Das erleichtert es Anwendungsschichtprogrammen, welche Clientzertifikate bereits unterstützen, auch FIDO2-authentifizierte Nutzer zu akzeptieren. Die bekannte Implementation OpenSSL und auch das später verwendete tlslite-ng machen im Falle einer erfolgreichen Authentifizierung der Anwendungsschicht des Servers das vollständige Clientzertifikat zugänglich. Dies soll auch nach erfolgreicher FIDO2-Authentifizierung passieren. Dafür muss der Server ein Clientzertifikat für den authentifizierten Nutzer erstellen oder ein bei der Registrierung erstelltes Zertifikat aus dem Speicher abrufen. In dem Zertifikat MUSS die Organisation (O) des Subjekts auf „fido2_authentication_cert“ gesetzt werden und die Organisationseinheit (OU) auf den FIDO2-Modus als String. Die ID eines Nutzers MUSS als Hexadezimalstring im State/ Province (ST)-Feld des Zertifikats gespeichert werden und der Nutzername, falls vorhanden, als Common Name (CN). Auf diese Weise kann der Nutzer identifiziert und die Authentifizierungsmethode festgestellt werden, ohne die Kompatibilität zu bereits bestehenden Programmen einzuschränken. In zukünftigen Versionen könnte auch eine Subject Alternative Name (SAN)-Erweiterung im Clientzertifikat verwendet werden, um Informationen an die Anwendungsschicht weiterzuleiten.
Ein Client, welcher in TLS 1.3 Daten zur Authentifizierung an den Server übermittelt, kann sich nicht sicher sein ob, der Server ihn anschließend als authentifiziert ansieht. Dies muss er separat über die Anwendungsschicht erfragen [42]. Im Fall des TFE-Handshakes kann ein *Silent Reject* eintreten, wenn der Server den Client nach der Risikoberechnung als nicht authentifiziert ansieht, den Handshake jedoch nicht abbricht. Für den Client ist dieser Fall nicht von einem TFE-Handshake mit erfolgreicher Authentifizierung zu unterscheiden. Der Server kann dem Client den Misserfolg der Authentifizierung anhand anwendungsabhängiger Mechanismen über die Anwendungsschicht mitteilen oder ihm stattdessen lediglich öffentlich zugängliche Bereiche und Daten der Anwendung verfügbar machen. Probleme können sich beim Caching ergeben. Angenommen ein Client synchronisiert nach jedem erfolgreichen Verbindungsaufbau alle Daten, welche der Server ihm zur Verfügung stellt, mit einem lokalen Verzeichnis. Im Falle eines Silent Rejects, welchen er nicht von einer erfolgreichen Authentifizierung unterscheiden kann, synchronisiert der Client das Verzeichnis mit den Daten, welche der Server jedem unauthentifiziertem Kommunikationspartner zugänglich macht oder welche ihn über den Misserfolg der Authentifizierung informieren sollen. Dabei werden sämtliche nutzerspezifischen Daten aus dem lokalen Verzeichnis gelöscht. Bei der nächsten erfolgreichen Authentifizierung müssen all diese Daten wieder neu ins Verzeichnis geladen werden.

5.3.5 PSK-Modus und 0-RTT-Daten

PSKs erlauben es, den Sicherheitskontext einer Verbindung an eine frühere Verbindung zu binden. Die Authentifizierung des Servers finden über den PSK statt und es werden keine Zertifikate übermittelt [42]. Für den ersten Handshake des TFE-Doppelhandshakes

KANN ein Server die Verwendung eines PSK erlauben. Dies reduziert die zu übertragene Datenmenge. Für den zweiten Handshake im FN-Modus, respektive den Handshake im FI-Modus, DARF ein Server einen PSK NICHT erlauben, da die Sicherheitseigenschaften der authentifizierten Verbindung nicht von denen einer möglicherweise unauthentifizierten Verbindung abhängen dürfen. Die Übermittlung von 0-RTT-Daten durch den Client ist somit nicht möglich, da dafür die Verwendung eines PSK notwendig ist [42]. Ein Server KANN die Verwendung von PSKs im TFE-Handshake auch gänzlich verbieten. Über die FIDO2-Erweiterung soll eine Authentifizierung pro Verbindung und nicht pro Sitzung stattfinden. Ein Server DARF daher NICHT das nach der FIDO2-Authentifizierung generierte Zertifikat zwischenspeichern und in einer späteren Verbindung mit PSK wiederverwenden. Implementationen, welche dies für Clientzertifikate des TCC-Handshakes tun, MÜSSEN daher, bevor sie ein zwischengespeichertes Zertifikat an die Anwendungsschicht weiterleiten, überprüfen, dass die Organisation des Subjekts nicht auf „fido2_authentication_cert“ gesetzt ist.

5.3.6 Verbesserungen

Ein großer Nachteil des doppelten Handshakes im FN-Modus ist die erhöhte Latenz. Durch den separaten Auf- und Abbau einer TCP- und TLS-Verbindung werden insgesamt drei zusätzliche Round Trip Times gegenüber dem gewöhnlichen TLS-1.3-Handshake benötigt. Dieses Problem lässt sich jedoch mit simplen Mitteln lösen. Der doppelte Handshake ist nötig, um zunächst einen ephemeren Nutzernamen zu etablieren und anschließend einen vollständigen Handshake inklusive FIDO2-Authentifizierung auszuführen, bei dem sich der Nutzer anhand des ephemeren Nutzernamens authentisiert. Wird das Paar aus ephemerem und Langzeit-Nutzernamen jedoch zu einem früherem Zeitpunkt auf dem Server gespeichert, so entfällt die Notwendigkeit für den ersten Handshake. Der Client kann in diesem Fall den ephemeren Nutzernamen direkt in der FIDO2ClientHelloExtension übertragen und mit dem zweiten Handshake beginnen. Dieses Ziel kann erreicht werden, indem Client und Server in einer vorherigen Verbindung einen ephemeren Nutzernamen für den nächsten Handshake generieren. Die für die Generierung notwendigen zufälligen Bytesequenzen von Client und Server können in einem vorherigen TFE-Handshake in den FIDO2AssertionRequest- und FIDO2AssertionResponse-Nachrichten übertragen werden. Der auf diese Weise modifizierte Handshake erfüllt somit zwei Funktionen: Einerseits authentifiziert er den Nutzer und andererseits etabliert er den neuen ephemeren Nutzernamen für den nächsten Handshake. Er übernimmt also die Aufgabe des zweiten Handshakes für die aktuelle Verbindung und die Aufgabe des ersten Handshakes für die nächste Verbindung. Dadurch muss in dem Fall, dass Server und Client in der Vergangenheit bereits miteinander kommuniziert haben, ein ganzer Handshake weniger ausgeführt werden. Für den aktualisierten TFE-FN-Handshake werden somit genau so viele RTTs benötigt wie für den Handshake im FI-Modus und den gewöhnlichen TLS-Handshake. Er wird im Folgenden als einfacher oder verkürzter TFE-FN-Handshake bezeichnet.

Für Server, welche diese Funktion anbieten, ist es sinnvoll, ephemere Nutzernamen über mehrere Tage zu speichern. Dadurch kann Tage – statt nur wenige Minuten – nach dem letzten Kontakt zwischen Server und Client noch ein einfacher anstatt einem doppelten Handshake ausgeführt werden. Maximal darf ein Server einen Eintrag für sieben Tage speichern. Auch unter Verwendung des einfachen TFE-FN-Handshakes ist keine Synchronisation zwischen verschiedenen Geräten nötig, da mehrere ephemere Nutzernamen auf den gleichen Langzeit-Nutzernamen abgebildet werden können. Verwendet ein Nutzer mehrere Geräte, so kann für jedes ein individueller ephemerer Nutzername zur späteren Verwendung gespeichert werden, ohne dass dieser an andere Geräte übermittelt werden muss.

5.4 Proof-Of-Concept-Implementation

Anhand der Proof of Concept (PoC)-Implementation wird die Umsetzbarkeit und die Funktionstüchtigkeit der FIDO2-Erweiterung bewiesen. Sie ermöglicht es, den TFE-Handshake zwischen Server und Client nicht nur theoretisch zu modellieren, sondern auch praktisch auszuführen und zu testen. Der Fokus liegt darauf, die Realisierung der Konzepte zu zeigen, und nicht darauf, eine sichere Implementation zur großflächigen Anwendung zu schaffen. Die PoC-Implementation zeigt dennoch sicherheitskritische Aspekte bei der Umsetzung der FIDO2-Erweiterung unter praxisnahen Bedingungen auf.
Als Grundlage für die Implementation dient das *tlslite-ng*-Projekt in der Version 0.8.0-aplha26 [29]. Dabei handelt es sich um eine Python-Implementation des TLS-Standards bis einschließlich Version 1.3. Das Projekt wurde wegen seiner guten Erweiterbarkeit ausgewählt, welche durch den simplen Aufbau der bereitgestellten Bibliotheken und Skripte sowie die einfache Installation und Bedienung ermöglicht wird. FIDO2-Funktionalität wird mit Hilfe der *python-fido2*-Bibliothek in Version 0.5.0 [53] umgesetzt. Der Quellcode der PoC-Implementation sowie eine Einführung in die Installation, Anwendung und Funktionsweise ist in [10] zu finden. Für die Verwendung wird Python3 benötigt.
Im Folgenden werden vorhandene Funktionen und Mechanismen sowie bestehende Limitierungen der Implementation genannt und implementationsabhängige Eigenschaften der FIDO2-Erweiterung erläutert. Anhand von Minimalbeispielen für die Verwendung der PoC-Implementation auf Server- und Clientseite wird die Registrierung und Authentifizierung von Nutzern gezeigt und verdeutlicht, wie bestehende Anwendungen zur Unterstützung der FIDO2-Erweiterung modifiziert werden können. Abschließend werden durchgeführte Tests und die Lizenz der Implementation beschrieben.

5.4.1 Funktionen und Limitierungen

Die Proof-Of-Concept-Implementation erfüllt die Anforderungen der in 5.3 beschriebenen Spezifikation bis auf wenige Ausnahmen. Sie stellt Funktionen bereit, welche es ermöglichen, den TFE-Handshake auf Client- und Serverseite durch wenige Parameter zu konfigurieren

und auszuführen. Des Weiteren beinhaltet sie Skripte, welche diese Methoden exemplarisch nutzen und deren Verwendung veranschaulichen. Es können alle optionalen Bestandteile der FIDO2-Authentifizierung zwischen Server und Client ausgetauscht werden und Werkzeuge zur Registrierung von Nutzern werden bereit gestellt. Es wird der TFE-Handshake im FN- und FI-Modus unterstützt. Neben dem doppelten Handshake erlaubt die Implementation das Generieren eines ephemeren Nutzernamens in einem früheren Handshake zur Unterstützung des einfachen TFE-FN-Handshakes. Welche Varianten des Handshakes vom Server angeboten werden sollen, kann bei dessen Konfiguration bestimmt werden.
Die Sicherheitseigenschaften der Implementation werden durch einige Limitierungen eingeschränkt. In tlslite-ng werden Zertifikate nicht verifiziert [29]. Dies ist jedoch notwendig, um serverseitige Endpunktauthentifizierung zu ermöglichen [42]. Clients können sich daher nicht sicher sein, dass sie mit dem beabsichtigten Endpunkt kommunizieren, was für eine sichere Ausführung des TFE-Handshakes notwendig ist. Wenige Einschränkungen entstehen durch die Verwendung der python-fido2-Bibliothek, welche nicht in jedem Anwendungsfall alle optionalen Parameter berücksichtigt, die Sicherheit der Implementation insgesamt jedoch nur geringfügig einschränkt.
Zur Vereinfachung werden an diesem Punkt lediglich externe USB-Authentifikatoren wie Sicherheitstokens unterstützt. Anstatt zu testen, ob die RP-ID einer Anfrage einem registrierbaren Domain-Suffix der Domain des RP-Servers entspricht, wird lediglich auf Gleichheit getestet. Für alle vom Server übertragenen WebAuthn-Erweiterungen wird angenommen, dass es sich um Authentifikator-Erweiterungen handelt, welche keiner weiteren Verarbeitung durch den Client bedürfen.
Aufgrund der beschriebenen Mängel sollte die PoC-Implementation nicht für sicherheitskritische Zwecke verwendet werden. Dennoch zeigt sie, dass die in 5.3 beschriebenen Spezifikationen unter realen Bedingungen umgesetzt werden können. Die Limitierungen können in großen Teilen durch die Verwendung etablierterer Bibliotheken umgangen werden.

5.4.2 Implementationsabhängige Eigenschaften der FIDO2-Erweiterung

Ausführung der RP-Server-Funktionalität

Die Spezifikation erlaubt, dass FIDO2-Funktionalität von der TLS-Serveranwendung selbst oder von einer separaten Anwendung ausgeführt wird. Die Ausführung einer solchen Anwendung kann lokal auf dem Server oder auf einem entfernten System stattfindet. In der PoC-Implementation werden alle notwendigen FIDO2-Operationen direkt in der TLS-Serveranwendung ausgeführt. Dies hat den Vorteil, dass kein Overhead durch die Kommunikation zu einer anderen Anwendung entsteht und die Implementation simpel gehalten werden kann. Für eine bessere Trennung verschiedener Komponenten, die Optimierung der Lastverteilung oder schnellere Wartung und Erweiterbarkeit kann jedoch auch die Umsetzung in einer separaten Anwendung von Vorteil sein.

Schutz des Registrierungsstatus

Laut Spezifikation sollten serverseitige Implementationen im TFE-FN-Handshake Mechanismen integrieren, welche es erschweren, die Anfrage in einer FIDO2AssertionRequest-Nachricht an einen registrierten Nutzer von der an einen nicht-registrierten Nutzer zu unterscheiden. Im FI-Modus wird die Anfrage nutzerunabhängig generiert, weshalb keine solchen Maßnahmen notwendig sind. Im FN-Modus jedoch wird für jeden registrierten Nutzer in den „allowCredentials“ eine Liste mit IDs auf ihn registrierter PKCSs übermittelt. Für einen unregistrierten Nutzer sind auf dem Server keine Informationen über PKCSs gespeichert. Die Liste der „allowCredentials“ wäre für ihn somit leer oder nicht vorhanden. Dadurch könnte diese als Erkennungsmerkmal für den Registrierungsstatus eines Nutzers verwendet werden. Einem Angreifer wäre es möglich, den Handshake für einen beliebigen Nutzer bei einem gegebenem Onlinedienst zu initiieren und anhand der Anfrage zu erkennen, ob dieser bei dem Dienst registriert ist. Nicht für alle Dienste mögen Gegenmaßnahmen relevant sein. Herauszufinden ob eine Person, beziehungsweise ein bestimmter Name, bei Facebook registriert ist, ist eine der Kernfunktionen der Anwendung. Ob die E-Mail-Adresse einer Kollegin oder eines Kollegen bei einer Webseite für Online-Glücksspiel registriert ist, sollte hingegen nicht ohne Weiteres in Erfahrung zu bringen sein.
Da sie für den grundlegenden Ablauf der Authentifizierung irrelevant sind, setzt die PoC-Implementation keine Gegenmaßnahmen um. Eine Möglichkeit könnte jedoch darin bestehen, den Server für unregistrierte Nutzer zufällig gewählte Parameter generieren zu lassen. Bei mehreren Anfragen für den gleichen Nutzernamen müssen diese gleich bleiben oder lediglich plausible Unterschiede aufweisen. Da es sehr unwahrscheinlich ist, dass die Liste der „allowCredentials“ oder andere Bestandteile für einen registrierten Nutzer in zwei aufeinanderfolgenden Anfragen gänzlich voneinander verschiedene Einträge beinhalten, wäre ein unregistrierter Nutzer ansonsten an inkonsistenten Parametern zu erkennen. Um für den gleichen Nutzernamen, auch falls dieser nicht registriert ist, stets die gleichen Anfrageargumente zu generieren, kann ein Server den Nutzernamen zusammen mit einem nur ihm bekannten Geheimnis als Seed eines Zufallsgenerators verwenden. Auf diese Weise berechnet der Zufallsgenerator stets die gleichen Werte. Um den Registrierungsstatus eines Nutzernamens nicht nach außen preiszugeben, muss ein Server im ersten Handshake des FN-Modus also zunächst alle an ihn übermittelten Paare aus ephemeren und Langzeit-Nutzernamen speichern, um im zweiten Handshake gegebenenfalls anhand des Nutzernamens mit einem deterministischen Verfahren Anfrageparameter zu erzeugen.
Der zusätzliche Speicher- und Rechenaufwand erhöht die Anfälligkeit des Servers für DoS-Angriffe. Zur Begrenzung des Speicherbedarfs kann ein Server alle oder die ältesten Paare aus ephemeren und Langzeit-Nutzernamen löschen, falls die Gesamtzahl der Einträge zu groß wird. Füllt ein Angreifer die Tabelle mit Paaren, so werden unter Umständen Einträge von legitimen Nutzern gelöscht. Dies könnte verhindern, dass der verkürzte TFE-FN-Handshake ausgeführt werden kann. Da der ephemere Nutzername für den doppelten Handshake jedoch nur sehr kurz gespeichert werden muss, ist es unwahrscheinlich,

dass ein Angreifer die Tabelle des Servers schnell genug mit Paaren füllen könnte, um legitime Anmeldungen gänzlich zu verhindern. Mit einigen Modifikationen am Handshake wäre es auch denkbar, dem Server das Auslagern verschlüsselter Informationen in einen clientseitig gespeicherten ephemeren Nutzernamen zu ermöglichen. Weitere Maßnahmen gegen DoS-Angriffe sollen an dieser Stelle nicht besprochen werden und werden von der Implementation nicht umgesetzt.

Zertifikatgenerierung
Bei erfolgreicher Authentifizierung muss die Implementation ein Clientzertifikat an die Anwendungsschicht übergeben. Dies kann nach jeder Authentifizierung neu generiert werden oder einmalig bei der Registrierung des Nutzers und zusammen mit dessen Nutzerinformationen gespeichert werden. Die PoC-Implementation generiert das Zertifikat bei der Registrierung des Nutzers und ruft es nach erfolgreichem TFE-Handshake aus dem Speicher ab. Dies erhöht den Speicherbedarf des Servers, reduziert jedoch den Rechenaufwand pro Authentifizierung. Beim Erstellen wird das Zertifikat vom Server signiert, um dessen Validität zu bescheinigen. Anstatt jedes Zertifikat mit dem gleichen Schlüssel zu signieren, wäre es denkbar, für jeden Hersteller von FIDO2-Authentifikatoren eine eigene Certificate Authority (CA) zu nutzen. Für Zertifikate, welche die Authentifizierung von Nutzern mit Authentifikatoren unterschiedlicher Hersteller bescheinigen, würde der Server zur Signatur also unterschiedliche Schlüssel verwenden. Auf diese Weise kann der Anwendungsschicht der Hersteller des Authentifikators mitgeteilt werden. Außerdem wird es Anwendungen erleichtert, die Unterstützung von Authentifikatoren auf eine bestimmte Menge von Herstellern einzuschränken. Vertraut eine Anwendung einem Hersteller nicht oder nicht mehr, so kann sie Zertifikate ablehnen, welche von der entsprechenden CA signiert wurden, zum Beispiel indem sie diese auf eine interne Certificate Revocation Liste setzt oder mit einer öffentlichen Liste vergleicht. Bestehende Mechanismen könnten auf diese Art ausgenutzt werden. Neben den in der Spezifikation festgelegten Pflichtfeldern könnten weitere Felder der Clientzertifikats genutzt werden, um unterschiedliche Informationen, wie beispielsweise die AAGUID des verwendeten Authentifikators, an die Anwendungsschicht weiterzureichen.

5.4.3 Registrierung

Die Registrierung von Nutzern ist kein Bestandteil der in dieser Arbeit vorgestellten FIDO2-Erweiterung. Für die Authentifizierung im TFE-Handshake ist die vorherige Registrierung jedoch erforderlich. Funktionen zur Nutzerverwaltung durch den Server werden daher über das Skript „fido2_server.py“ bereit gestellt. Dieses erlaubt es, eine Nutzerdatenbank anzulegen, die Generierung einer neuen PKCS auf einem Authentifikator anzustoßen und alle notwendigen Informationen über einen Nutzer und die mit ihm assoziierten PKCSs in der Datenbank zu sichern. Die Datenbank kann anschließend an den TLS-Server übergeben werden, um die Authentifizierung der gespeicherten Nutzer im TFE-Handshake zu ermöglichen. Die Registrierung muss also lokal auf dem Server stattfinden oder die

```
> fido2_server.py setup --db-path fido2.db

> fido2_server.py register --db-path fido2.db --rp-id
   ↪ localhost --cert maxmustermann_cert.pem

> fido2_server.py register --db-path fido2.db --name
   ↪ erikamusterfrau --display-name "Erika Musterfrau" --rp-
   ↪ id localhost --server-cert server_cert.pem --server-key
   ↪  server_key.pem
```

Abbildung 5.7: Beispielaufrufe des „fido2_server.py“-Skripts
Quelle: Eigene Abbildung

Datenbank dem Server anderweitig zugänglich gemacht werden.
In Abbildung 5.7 werden exemplarisch Aufrufe an das Skript gezeigt, welche dessen Funktionsweise verdeutlichen sollen. Durch den ersten Aufruf wird eine Nutzerdatenbank in der Datei „fido2.db“ angelegt. Der Zweite Aufruf registriert den Nutzer Max Mustermann. Dafür wird sein Nutzername, sein Anzeigename, die RP-ID des Dienstes, für welchen er sich registrieren möchte, und sein Clientzertifikat übergeben. Der Authentifikator, welcher für die Registrierung verwendet werden soll, muss zu Beginn der Ausführung des Skriptes verfügbar sein. Im letzten Aufruf wird Erika Musterfrau registriert. Da für sie noch kein Clientzertifikat existiert, werden stattdessen das Zertifikat und der private Schlüssel des Servers übergeben, mit Hilfe derer das Skript ein valides Clientzertifikat erstellen kann. Dafür wird das Skript „generate_client_certificate.sh“ aufgerufen, welches die Generierung des Clientzertifikates mittels OpenSSL-Werkzeugen übernimmt. Sind diese auf dem ausführenden System nicht vorhanden, muss das Clientzertifikat auf andere Weise generiert und anschließend an das Registrierungsskript übergeben werden. Über den Parameter `--encryption-key server_key.pem` , welcher bei der Generierung der Datenbank übergeben werden kann und, falls vorhanden, bei allen späteren Operationen ebenfalls gesetzt werden muss, kann die Nutzerdatenbank verschlüsselt werden. Weitere Beispielaufrufe des Skripts sind in der „README.md“-Datei von [10] zu finden.

Abbildung D.3 des Anhangs zeigt das durch den ersten Aufruf generierte Datenbankschema der Nutzerdatenbank. Für jeden Nutzer wird in der Relation *users* dessen ID und, sofern vorhanden, auch der Nutzername und ein Anzeigename gespeichert. In WebAuthn müssen Authentifizierungs- und Autorisierungsentscheidungen basierend auf der ID eines Nutzers geschehen [4]. Um die Eindeutigkeit eines Nutzernamen zu garantieren, werden alle Nutzernamen gemäß RFC 8266 [45] verarbeitet, bevor sie in der Datenbank gesichert werden. Anschließen kann im Authentifizierungsprozess die Nutzer-ID über den gleichermaßen verarbeiteten Nutzernamen abgefragt und für zukünftige Operationen genutzt werden. Wie in [4] empfohlen, besteht die Nutzer-ID aus 64 zufällig gewählten Bytes und enthält keine Informationen wie Nutzername oder E-Mail-Adresse des Nutzers.

In der Relation *credentials* werden öffentliche Informationen über eine PKCS, den Authentifikator, an welchen sie gebunden ist, und die ID des Nutzers, welchem sie zugeordnet ist, gesichert. Dabei ist es möglich, mehrere Einträge für den gleichen Nutzer zu speichern. Dadurch können auf einen Nutzer mehrere FIDO2-Geräte registriert werden. Eine PKCS wird anhand der eindeutigen ID identifiziert. Neben dieser wird ihr öffentlicher Schlüsselanteil und die AAGUID des verwendeten Authentifikators gespeichert. Auch der FIDO2-Modus, in welchem eine PKCS verwendet werden darf, wird als Integer codiert gesichert.
Die *certificates*-Relation speichert die einem Nutzer zugeordneten Zertifikate. Falls ein Server verschiedene FIDO2-Modi für den gleichen Nutzer erlaubt, kann an dieser Stelle pro Nutzer und Modus ein Clientzertifikat vorhanden sein.
Die Relation *eph_user_names* wird benötigt, um ephemere Nutzernamen auf Langzeit-Nutzernamen abzubilden. Anstatt des Langzeit-Nutzernamens wird an dieser Stelle direkt die Nutzer-ID gespeichert, da diese für alle späteren Operationen verwendet wird. Neben dem Paar aus ephemerem Nutzernamen und ID wird darüber hinaus der Zeitpunkt gespeichert, bis zu welchem der Eintrag gültig ist. Ein Eintrag wird gelöscht, falls der ephemere Nutzername in einem Handshake verwendet wurde oder seine Gültigkeit abgelaufen ist.

5.4.4 Authentifizierung

Clientseitige Nutzung der Bibliothek

Funktionen zur Ausführung des TFE-Handshakes wurden der tlslite-ng Bibliothek hinzugefügt, respektive bestehende Funktionen modifiziert. Die neu definierte Funktion `TLSConnection.handshakeClientFIDO2()` ermöglicht die clientseitige Ausführung des TFE-Handshakes und stellt eine Alternative zu bereits bestehenden Handshake-Funktionen dar, welche einen anonymen, zertifikat- oder SRP-basierten Verbindungsaufbau erlauben. In Anhang D.1 wird ein Minimalbespiel für deren Verwendung in einem HTTPS-Client gezeigt. Zur Unterstützung der FIDO2-Erweiterung im FI-Modus muss der Funktion lediglich der Domain Name des Servers übergeben werden. Für den FN-Modus wird zusätzlich der zu verwendende Nutzername übergeben. Stattdessen kann für diesen Modus auch der Speicherort eines zuvor generierten ephemeren Nutzernamen übergeben werden sowie optional der eines in diesem Handshake für eine spätere Verbindung erzeugten ephemeren Namens. Weitere Parameter beinhalten ein Objekt zur Kommunikation mit einem Nutzer, zum Beispiel um diesen zu einer Interaktion aufzufordern, und ein Event, welches genutzt werden kann, um die Authentifizierung abzubrechen. Im Minimalbeispiel wird der FN-Modus verwendet, ohne einen vorab vereinbarten ephemeren Nutzernamen abzurufen oder einen solchen für einen späteren Verbindungsaufbau zu speichern. Es wird also der vollständige TFE-FN-Handshake ausgeführt. Um stattdessen den FI-Modus zu verwenden, reicht es aus, keinen Nutzernamen zu übergeben. Da keine passenden Objekte definiert werden, wird das Standardverfahren zur Nutzerkommunikation und zum Abbruch der Authentifizierung über die Kommandozeile gewählt. Nachdem die Verbindung etabliert ist, sendet der Democlient einen GET-Request an den Server und empfängt das Ergebnis.

Verfahren	Speichervariable innerhalb des Session-Objektes	Wert bei Erfolg	Wert bei Misserfolg	Weitere Eigenschaften
Clientzertifikat	clientCertChain	valides X509CertChain Objekt	None	–
FIDO2	clientCertChain	valides X509CertChain Objekt	None	„fido2_authentication_cert“ als Organisation des Subjekts
SRP	srpUserName	String	None	–

Abbildung 5.8: tlslite-ng Authentifizierungsverfahren im Überblick
Quelle: Eigene Abbildung. Enthält Informationen aus „README.md“ in [29]

Serverseitige Nutzung der Bibliothek

Die serverseitige Ausführung des TFE-Handshakes wurde der bereits in tlslite-ng vorhandenen `TLSConnection.handshakeServer()`-Methode hinzugefügt, über welche der Server alle unterstützen Handshake-Varianten ausführt. Abbildung D.2 des Anhangs zeigt ein Minimalbeispiel für einen HTTPS-Server mit Unterstützung für die FIDO2-Erweiterung. Um den TFE-Handshake zu ermöglichen, müssen der Handshake-Methode die `fido2_params` übergeben werden. Dabei handelt es sich um ein Dictionary mit mehreren Bestandteilen. Mindestens der Pfad zu einer Nutzerdatenbank und die RP-ID des Servers müssen gesetzt werden. Falls die Datenbank verschlüsselt gespeichert wird, muss des Weiteren der passende Schlüssel übergeben werden. Optionale Parameter beinhalten eine Flag, welche angibt, ob FIDO2-Authentifizierung von jedem Nutzer gefordert wird, und eine weitere, welche signalisiert, ob der ephemere Nutzername für die nächste Verbindung in diesem Handshake generiert werden soll. Sollen nicht alle FIDO2-Modi angeboten werden, so kann eine Liste der gewünschten Modi übergeben werden. Die Argumente `certChain` und `privateKey` ermöglichen es dem Server, sich gegenüber dem Client zu authentifizieren und sind standardmäßig in der Handshake-Methode enthalten. Der Server im Minimalbeispiel bietet FIDO2 im FI- und im vollständigen und verkürzten FN-Modus an. Er verwendet eine unverschlüsselte Nutzerdatenbank und setzt den Handshake auch dann fort, wenn der Client keine FIDO2-Authentifizierung anbietet.

Auf `TLSConnection.session.clientCertChain` kann nach einem Handshake von einer Serveranwendung zugegriffen werden. Falls die FIDO2-Authentifizierung erfolgreich war, ist an dieser Stelle eine Zertifikatskette verfügbar. Das letzte Glied der Kette stellt dabei Informationen über den Nutzer, welcher authentifiziert wurde, bereit. Diese sind wie in 5.3 gefordert gesetzt. Die Tabelle in Abbildung 5.8 zeigt die Informationen, welche nach unterschiedlichen Authentifizierungsverfahren in `TLSConnection.session` gespeichert werden. Neben dem Variablennamen innerhalb des Session-Objekts werden der Wert bei erfolgreicher und unerfolgreicher Authentifizierung sowie weitere Eigenschaften der Variable angegeben.

Die Ausgabe des Demoservers besteht bei einem erfolgreichen TFE-Handshake mit dem zuvor besprochenen Democlient aus drei Zeilen. Zuerst erfolgt eine Meldung darüber, dass eine Verbindung beendet wurde, ohne Anwendungsdaten zu übertragen – dies ist die Verbindung, welche für den ersten Handshake des doppelten Handshakes benötigt wird. Anschließend wird der Nutzername des authentifizierten Nutzers ausgegeben und zuletzt die Information über den vom Nutzer gesendeten „GET“-Request.

5.4.5 Von Clientzertifikaten zu FIDO2: Das „tls.py“-Skript

Das in tlslite-ng enthaltene „tls.py“-Skript erlaubt per Kommandozeile das Starten eines HTTPS-Servers oder -Clients und gibt nach erfolgreichem Verbindungsaufbau Informationen über diesen aus. Es unterstützt die Verwendung von Clientzertifikaten. Wird ein Handshake unter Nutzung dieser ausgeführt, so wird anschließend ein SHA1-Hash über die Bytes des Zertifikates ausgegeben. Andernfalls wird angegeben, dass kein Clientzertifikat übertragen wurde. Das Skript wurde modifiziert, um mit Hilfe der Bibliotheksfunktionen auch die FIDO2-Erweiterung zu unterstützen. Die Verwendung der Funktionen erfolgt dabei ähnlich den Minimalbeispielen. Da nach erfolgreicher FIDO2-Authentifizierung das Clientzertifikat an der gleichen Stelle gespeichert wird wie nach einem clientzertifikatbasierten Handshake, wird die Verwendung der clientseitigen Authentifizierung ohne Änderungen an bestehenden Mechanismen angezeigt. Durch Überprüfen der Organisation des Subjekts des Zertifikates werden beide Fälle voneinander unterschieden und im Falle von FIDO2-Authentifizierung zusätzlich der verwendete Handshake-Modus ausgegeben.
Abbildung 5.9 zeigt Beispielaufrufe für die modifizierte Variante des Skripts. Dabei sollten die Aufrufe von separaten Kommandozeilen ausgeführt werden. Der erste Aufruf startet einen HTTPS-Server, welcher auf Anfragen wartet und den TFE-Handshake im FI-Modus und in voller und verkürzter Variante des FN-Modus unterstützt. Neben dem Zertifikat und dem privaten Schlüssel des Servers, welche immer für den Handshake übergeben werden müssen, bekommt das Skript den Pfad zu einer vorher angelegten Nutzerdatenbank mitgeteilt und die Anweisung den verkürzten TFE-FN-Handshake zuzulassen sowie detaillierte Informationen während des Verbindungsaufbaus auszugeben. Weitere Parameter steuern zusätzliche optionale Bestandteile der `TLSConnection.handshakeServer()`-Methode, wie zuvor beschrieben.

```
> tls.py server -k server_key.pem -c server_cert.pem --fido2-
  ↪ db fido2.db --pre-share-euname --verbose localhost:4443

> tls.py client --fido2 -u maxmustermann --eph-uname-out
  ↪ eph_user_name.bin --verbose localhost:4443
```

Abbildung 5.9: Beispielaufrufe des „tls.py“-Skripts
Quelle: Eigene Abbildung

Der zweite Aufruf startet einen Client, welcher sich mit dem HTTPS-Server unter Verwendung des vollen TFE-FN-Handshakes verbindet. Den ephemeren Nutzernamen für die nächste Verbindung speichert er in der Datei „eph_user_name.bin“. Auch die Clientanwendung wird über das `verbose`-Flag dazu aufgefordert, ausführliche Informationen über den Nachrichtenaustausch auszugeben. Für einen erfolgreichen Handshake muss der Nutzer zurvor registriert worden sein. Nach dem Ende des Verbindungsaufbaus zeigen Server und Client Informationen über dessen Parameter an. In der „README.md“-Datei von [10] sind auch für das „tls.py“-Skript weitere exemplarische Aufrufe vorhanden

5.4.6 Tests

Die Funktionsfähigkeit der PoC-Implementation wurde durch verschiedene Tests überprüft. Insbesondere wurden dafür der HTTPS-Server und -Client, welche durch das „tls.py“-Skript zur Verfügung gestellt werden, verwendet. Die volle Liste der Parameter, welche für unterschiedliche Tests verwendet werden können, wird bei einem Aufruf des Skriptes ohne Parameter angezeigt. Die Ausführung des Servers wurde dabei lokal und entfernt auf einem Raspberry Pi getestet. Als Authentifikator wurde ausschließlich der Yubico Security Key als externes Sicherheitstoken eingesetzt. Die Verwendung mehrerer Tokens im parallelen Betrieb wurde getestet. Manuelle Unittests wurden während des gesamten Entwicklungsprozesses vollzogen. Weitere Tests wurden automatisiert über das Skript„tlstest.py“ umgesetzt. Darin werden verschiedene Szenarien überprüft, in denen unter anderem böswillige Anfragen eines Servers und der Abbruch der Authentifizierung durch den Client berücksichtigt werden. Besonderer Wert wurde darauf gelegt, die Ausführbarkeit bereits vorhandener Tests nicht einzuschränken, um die Funktionstüchtigkeit aller in tlslite-ng vorhandenen Funktionen weiterhin zu überprüfen und Beeinflussung durch die FIDO2-Erweiterung auszuschließen.

5.4.7 Lizenz

Das *tlslite-ng*-Projekt steht unter der GNU LGPLv2. Die PoC-Implementation, welche tlslite-ng weiterentwickelt, soll unter GNU GPLv3 verfügbar sein. Die Umlizensierung von GNU LGPLv2 in GPLv2 oder neuere Lizenzversionen wird in der LGPLv2 ausdrücklich erlaubt [21]. Für die Verwendung des *python-fido2*-Projekts müssen dessen Copyright-Hinweise in die Lizenz übernommen werden [53]. Die entsprechenden Anmerkungen wurden daher neben bereits bestehenden in die Lizenz eingefügt. Die vollständige Lizenz ist in der Datei „LICENSE“ unter [10] verfügbar.

6 Fazit

Obwohl zahlreiche Probleme der Passwortauthentifizierung seit Ende der 70er Jahre bekannt sind und Alternativen existieren, stellt das Verfahren noch heute die mit Abstand verbreitetste Authentifizierungsform dar [28]. Während Nutzer Probleme damit haben, sich Passwörter zu merken [23], steigt die Zahl der Dienste, denen gegenüber sie sich in ihrem Alltagsleben authentifizieren müssen, an [28]. Als Folge sind teure Anrufe an Kundendienste nötig oder es muss auf teilweise noch unsichere Ersatzmechanismen zurückgegriffen werden [23]. 12,6 Minuten jede Woche, also 10,9 Stunden pro Jahr, verbrachten die für [39] befragten Personen im Durchschnitt mit der Eingabe und dem Zurücksetzen von Passwörtern.
Neben einer einfacheren Anwendung für Nutzer bieten die von der FIDO-Alliance veröffentlichen Verfahren eine erhöhte Sicherheit gegenüber der Passwortauthentifizierung [15]. Dennoch konnten weder diese noch andere Vorschläge Passwörter bisher als Standardverfahren ablösen. Dies könnte daran liegen, dass keiner der Mechanismen so schnell von Diensteanbietern zur Verfügung gestellt und von Nutzern verwendet werden kann wie Passwörter [44].
In dieser Arbeit wurde ein Vorschlag präsentiert, mit FIDO2 die aktuellen Spezifikationen der FIDO-Alliance über den Erweiterungsmechanismus in den TLS-1.3-Handshake einzubinden. Der Entwurf des neuen TFE-Handshakes stellt dabei die sichere Umsetzung des Authentifizierungsprozesses in den Fokus. Die Nachrichtenphasen orientieren sich am Ablauf des TLS-Handshakes unter Verwendung von Clientzertifikaten, genau wie die zur Kommunikation mit der Anwendungsschicht verwendeten Mechanismen. Dies verhindert einerseits Fehler in der Konzeption und in den Implementationen und erleichtert es andererseits clientzertifikatbasierten Anwendungen, die FIDO2-Erweiterung zu nutzen. Für die Verschlüsselung aller sensiblen Daten, die während des Handshakes übertragen werden, wird Forward Secrecy gewährleistet. Die Integration der FIDO2-Erweiterung ist nicht nur sicher, sondern auch effizient möglich. Zusätzlich zum doppelten Handshake wurde ebenfalls eine verkürzte Variante vorgestellt, welche es ermöglicht, ohne Einschränkung der Sicherheitseigenschaften den Verbindungsaufbau im FI und FN-Modus mit der gleichen Anzahl an RTTs durchzuführen wie bei einer clientseitig unauthentifizierten TLS-1.3-Verbindung, sollten Client und Server zuvor bereits miteinander kommuniziert haben. Neben dem Entwurf der Erweiterung wurde eine Implementation zur Verfügung gestellt, welche die Funktionstüchtigkeit des theoretischen Konzepts auch praktisch unter Beweis stellt. Trotz bestehender Sicherheitseinschränkungen, erlaubt es die Implementation, den doppelten und einfachen TFE-Handshake im FN-Modus sowie den TFE-FI-Handshake zwischen zwei Kommunikationspartnern auszuführen und nachzuvollziehen. Es wurde anhand eines Beispiels gezeigt, wie die FIDO2-Erweiterung durch simple Änderungen in einer Anwendung an Stelle von Clientzertifikaten verwendet werden kann und wie Server

und Client dafür konfiguriert werden können.
Um flächendeckend zur Anwendung zu kommen, müsste die FIDO2-Erweiterung standardisiert werden. Dazu müssten neu definierte Erweiterungs-, Nachrichten- und Altert-Typen einen Review-Prozess durchlaufen und in die von der IANA geführte Registrierungsdatenbank übernommen werden [46]. Nach der Standardisierung wäre die FIDO2-Erweiterung mit TLS Teil eines der meist genutzten Sicherheitsprotokolle im Internet [51]. Für Diensteanbieter wäre es auf diese Weise einfacher, FIDO2-Authentifizierung anstelle von Passwörtern zur Verfügung zu stellen. Die Authentifizierungsmethode könnte von allen bekannten Onlinediensten, aber auch von kleinen Anbietern ohne Know-How in diesem Bereich, durch nur wenige Änderungen bei der TLS-Konfiguration angeboten werden. Für Nutzer könnte die potentiell breitere Verfügbarkeit einen weiteren Anreiz darstellen, auf das kryptographisch sichere Verfahren umzusteigen. Trotz des weiten Weges, der nötig ist, um als Standard übernommen, von gängigen Implementationen integriert, von Onlinediensten angeboten und schließlich von Nutzern akzeptiert zu werden, birgt die FIDO2-Erweiterung somit insgesamt zumindest das Potenzial, einer passwortlosen Zukunft einen Schritt näher zu kommen.

Anhang A Erweiterungstypen

In den nachfolgenden Abbildungen wird die Größe eines Feldes in Bytes darunter angegeben. Für Felder variabler Länge wird der Wert als unsigned Big-Endian-Integer codiert in einem separaten Längenfeld übertragen. Ebenso wird die Anzahl der Einträge für Felder codiert, welche mehrfach auftreten können. Die Anzahl der Felder und die Länge von Feldern werden mit Großbuchstaben bezeichnet. Vorangestellte nullen zeigen an wie viele der höchstwertigen Bits auf null gesetzt sein müssen.

A.1 FIDO2ClientHelloExtension

Kennung: fido2_clienthello_extension = 55
Erlaubt in Nachrichten: ClientHello

flags

Typ: Bitstring
Länge: 1 Byte

0: e-Bit Gibt an ob das optionale Feld eph_user_name vorhanden ist.

1-3: Reserviert für zukünftige Verwendung.

4-7: Modus des Handshakes.

eph_user_name (optional)

Typ: Bytestring
Länge: 32 Bytes

In diesem Feld wird der ephemere Nutzername übertragen, welcher in einem vorherigen Handshake mit dem Server generiert wurde.

ClientHello-Erweiterung: FIDO2ClientHelloExtension

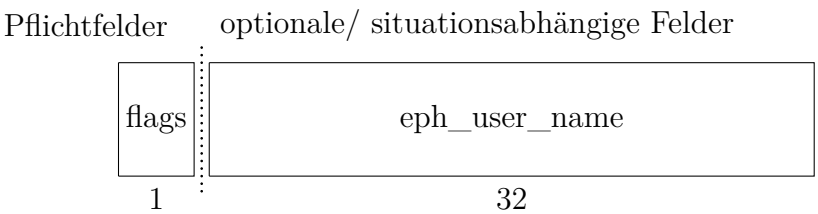

Abbildung A.1: Erweiterungsformat der FIDO2ClientHelloExtension
Quelle: Eigene Abbildung

Anhang B Nachrichtentypen

B.1 FIDO2NameRequest

Kennung: fido2_name_request = 26

eph_user_name_server_share

Typ: Bytestring

Länge: 32Bytes

In diesem Nachrichtenfeld werden 32 vom Server zufällig generierte Bytes übertragen, welche in die Generierung eines ephemeren Nutzernamens eingehen.

Server-Nachricht: FIDO2NameRequest

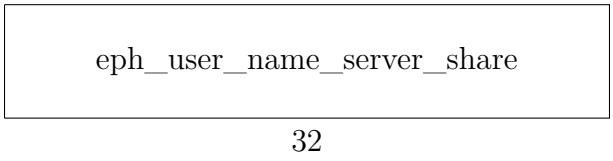

Abbildung B.1: Nachrichtenformat des FIDO2NameRequest
Quelle: Eigene Abbildung

B.2 FIDO2NameResponse

Kennung: fido2_name_response = 27

eph_user_name_client_share

Typ: Bytestring
Länge: 32 Bytes

In diesem Nachrichtenfeld werden 32 vom Client zufällig generierte Bytes übertragen, welche in die Generierung eines ephemeren Nutzernamens eingehen.

user_name

Typ: UTF-8 String
Länge: 0 bis $2^8 - 1$ Bytes + 1 Längen-Byte

In diesem Nachrichtenfeld wird ein Langzeit-Nutzername übertragen.

Client-Nachricht: FIDO2NameResponse

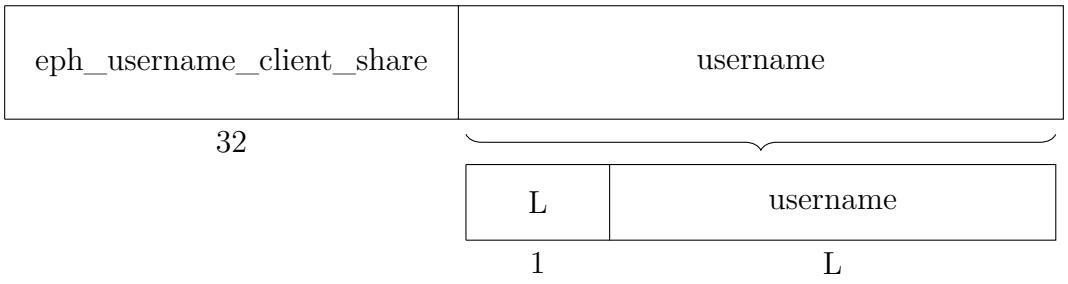

Abbildung B.2: Nachrichtenformat der FIDO2NameResponse
Quelle: Eigene Abbildung

B.3 FIDO2AssertionRequest

Kennung: fido2_assertion_request = 28

flags

Typ: Bitstring
Länge: 1 Byte

0: t-Bit Gibt an ob das optionale Feld „timeout“ vorhanden ist.

1: r-Bit Gibt an ob das optionale Feld „rp_id“ vorhanden ist.

2: u-Bit Gibt an ob das optionale Feld „user_verification“ vorhanden ist.

3: a-Bit Gibt an ob das optionale Feld „allow_credentials“ vorhanden ist.

4: e-Bit Gibt an ob das optionale Feld „extensions“ vorhanden ist.

5: n-Bit Gibt an ob das optionale Feld „eph_user_name_server_share“ vorhanden ist.

6-7: Reserviert für zukünftige Verwendung.

challenge

Typ: UTF-8 String
Länge: 16 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes

Dieses Feld beinhaltet die vom Server generierte Challenge, welche zusammen mit anderen Daten signiert wird, um eine Assertion zu generieren [4].

timeout (optional)

Typ: unsigned long (Big-Endian)
Länge: 8 Bytes

Das „timeout“ gibt die Zeit in Millisekunden an, die eine Relying Party bereit ist, auf eine Antwort zu warten. Der Wert darf vom Client überschrieben werden [4].

rp_id (optional)

Typ: UTF-8 String
Länge: 0 bis $2^8 - 1$ Bytes + 1 Längen-Byte

Die „rp_id“ ist die Kennung der Relying Party. Falls nicht vorhanden, wird der Domain Name des Servers verwendet [4].

allow_credentials (optional)

Typ: Zusammengesetzt
Länge: 0 bis $(2^6 - 1) * (2^{16} + 2^8 + 2 + (2^6 - 1) * 2^6)$ Bytes + 1 Byte für die Anzahl der Elemente

Dieser Eintrag beinhaltet eine Liste der für den RP-Server akzeptablen PKCSs. [4]. Es können maximal 63 Elemente übertragen werden.

id

Typ: Bytestring
Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes
Dieser Eintrag beinhaltet die eindeutige Kennung einer PKCS [4].

type

Typ: UTF-8 String
Länge: 0 bis $2^8 - 1$ Bytes + 1 Längen-Byte
Dieser Eintrag beinhaltet den Typ der PKCS. Aktuell muss der Wert auf „public-key“ gesetzt sein [4].

transports

Typ: Liste von UTF-8 Strings
Länge: 0 bis $(2^6 - 1) * (1 + 2^6 - 1) + 1$ Byte für die Anzahl der Elemente
Dieses Feld beinhaltet einen Hinweis darauf, wie der Client mit dem Authentifikator kommunizieren kann, welcher die PKCS speichert. Mögliche Werte sind „usb“, „nfc“, „ble“und „internal“ [4]. Die Liste kann 0 bis 63 Elemente enthalten.

user_verification (optional)

Typ: UTF-8 String
Länge: 0 bis $2^8 - 1$ Bytes + 1 Längen-Byte

Der Eintrag beschreibt die Vorgaben des Servers für die Nutzerverifikation. Der Eintrag kann den Wert „required“, „preffered“ und „discouraged“ annehmen. Je nachdem ist Nutzerverifikation vorgeschrieben, gewünscht falls das verwendete Gerät sie unterstützt oder darf nicht stattfinden [4].

extensions (optional)

Typ: Zusammengesetzt
Länge: 0 bis $(2^6 - 1) * (2^5 + 2^{16} + 1)$ Bytes + 1 Byte für die Anzahl der Elemente

Erweiterungen beinhalten zusätzliche Parameter, welche vom Client respektive dem Authentifikator bearbeitet werden müssen [4]. Die Werte können zu einer CBOR Map nach [6] mit der ID als Schlüssel und den Daten als Wert zusammengesetzt werden. Es können maximal 63 Elemente übertragen werden.

extension_id

Typ: UTF-8 String
Länge: 0 bis $2^5 - 1$ Bytes + 1 Längen-Byte
Dieser Eintrag beinhaltet die eindeutige Kennung der Erweiterung [4].

extension_data

Typ: JSON-codierbarer Wert. Von Erweiterung definiert.
Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes
Dieser Eintrag beinhaltet den Input für Clienterweiterungen [4].

eph_user_name_server_share (optional)

Typ: Bytestring
Länge: 32 Bytes

In diesem Nachrichtenfeld werden 32 vom Server zufällig generierte Bytes übertragen, welche in die Generierung eines ephemeren Nutzernamens für eine spätere Verbindung eingehen.

Nachrichtenlänge

Die maximale Länge l, welche ein FIDO2AssertionRequest annehmen kann, berechnet sich wie unten angegeben. Eine TLS-Handshake-Nachricht darf maximal $m = 2^{24} - 1$ Bytes beinhalten [42]. Da gilt $l < m$, überschreitet kein FIDO2AssertionRequest die Maximallänge für Handshake-Nachrichten.

$$
\begin{aligned}
& 1 \\
& 2^{16} + 1 \\
& 8 \\
& 2^8 \\
& 63 * (2^{16} + 2^8 + 2 + 63 * 2^6) + 1 \\
& 2^8 \\
& 63 * (2^{16} + 2^5 + 1) + 1 \\
+ \quad & 32 \\
\hline
& 8.595.977 \text{ Bytes} = l
\end{aligned}
$$

Server-Nachricht: FIDO2AssertionRequest

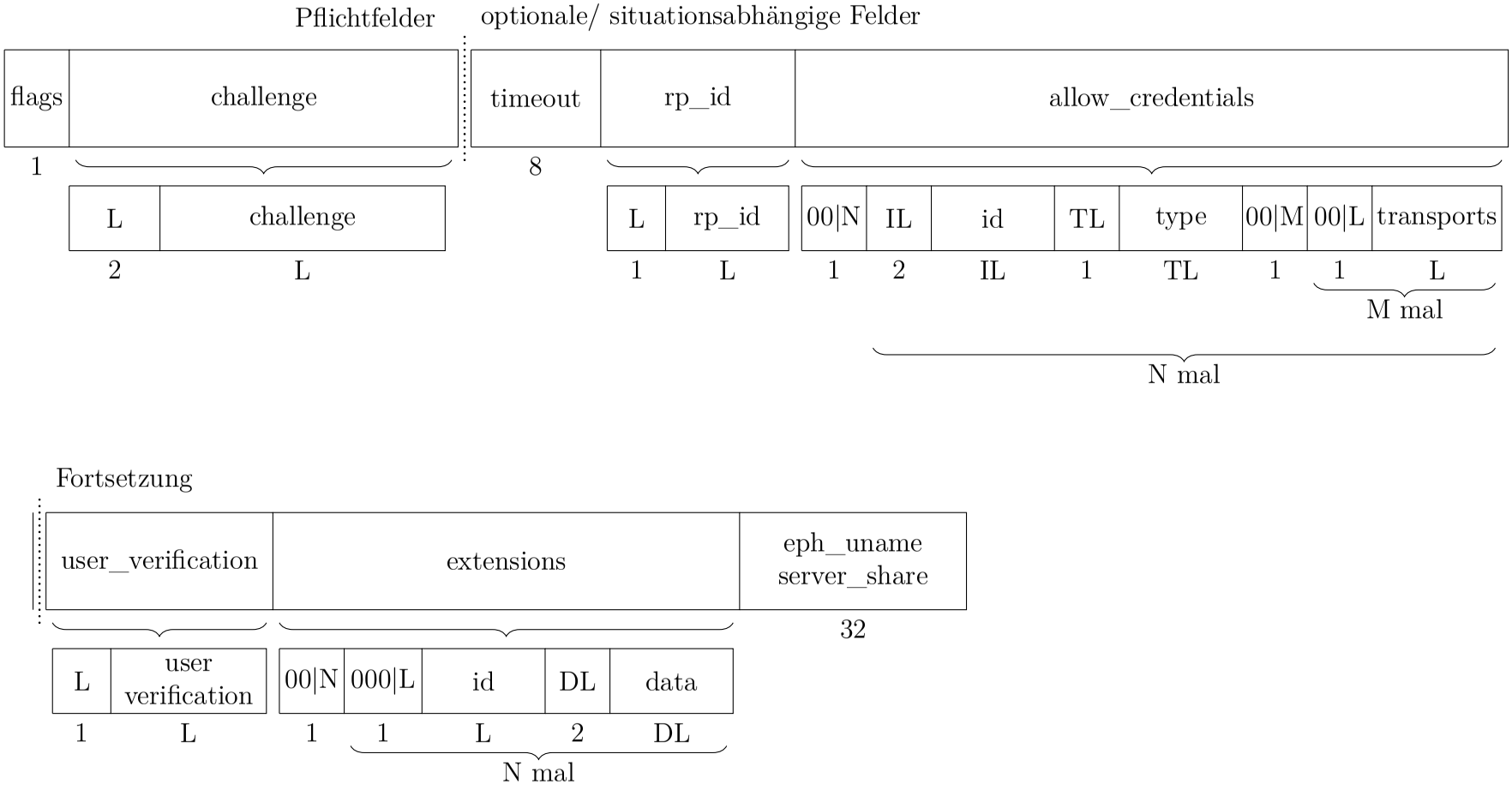

Abbildung B.3: Nachrichtenformat des FIDO2AssertionRequest
Quelle: Eigene Abbildung

B.4 FIDO2AssertionResponse

Kennung: fido2_assertion_response = 29

flags

Typ: Bitstring
Länge: 1 Byte

0: u-Bit Gibt an ob das optionale Feld „user_handle“ vorhanden ist.

1: s-Bit Gibt an ob das optionale Feld „selected_credential_id“ vorhanden ist.

2: e-Bit Gibt an ob das optionale Feld „client_extension_output“ vorhanden ist.

3: n-Bit Gibt an ob das optionale Feld „eph_user_name_client_share“ vorhanden ist.

4-7: Reserviert für zukünftige Verwendung.

client_data_json

Typ: JSON-serialisiertes „clientData“-Objekt
Länge: 0 bis $2^{18} - 1$ Bytes + 3 Längen-Bytes

Dieser Eintrag beinhaltet die JSON-serialisierte Form der „client_data“, welche dem Authentifikator für die Generierung der Assertion übergeben wurden. Die exakte Serialisierung muss beibehalten werden, da ein Hashwert darüber generiert wird [4].

authenticator_data

Typ: Zusammengesetzt
Länge: 37 bis $54 + 2 * (2^{16} + 1) + (2^6 - 1) * (2^5 + 2^{16} + 1)$ Bytes

Die „authenticator_data“ werden bei der Generierung einer Assertion vom Authentifikator zurück geliefert [4].

rp_id_hash

Typ: Bytestring
Länge: 32 Bytes
Das Feld beinhaltet den SHA-256-Hash der RP-ID, an welche die verwendete PKCS gebunden ist [4].

flags

Typ: Bitstring
Länge: 1 Byte

0: up-Bit Gibt an ob ein Test auf Nutzeranwesenheit erfolgreich war [4].

2: uv-Bit Gibt an ob eine Nutzerverifikation erfolgreich war [4].

6: at-Bit Gibt an ob das optionale Feld „attested_credential_data“ vorhanden ist [4].

7: ed-Bit Gibt an ob das optionale Feld „authenticator_extension_output“ vorhanden ist [4].

1, 3-5: Reserviert für zukünftige Verwendung [4].

sign_count

Typ: unsigned integer (Big-Endian)

Länge: 4 Bytes

Der „sign_count“ ist ein Zähler, welcher vom Authentifikator gespeichert und bei erfolgreicher Generierung einer Assertion inkrementiert wird [4].

attested_credential_data (optional)

Typ: Zusammengesetzt

Länge: 20 bis $16 + 2 * (2^{16} + 1)$ Bytes

Der Eintrag ist für die Authentifizierung nicht relevant, da er in den „authenticator_data“ lediglich auftaucht, falls ein Attestierungsobjekt erzeugt wird [4]. Er wird benötigt, falls auch die Registrierung von Nutzern über den TFE-Handshake möglich gemacht werden soll.

aaguid

Typ: Bytestring

Länge: 16 Bytes

Dieser Eintrag beinhaltet die eindeutige Kennung eines Authentifikatormodells [4].

credential_id

Typ: Bytestring

Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes

Dieser Eintrag beinhaltet die eindeutige Kennung einer PKCS [4].

public_key

Typ: COSE_Key-codierter öffentlicher Schlüssel, wie definiert in Abschnitt 7 von [47]

Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes

Dieser Eintrag beinhaltet den öffentlichen Schlüsselanteil einer PKCS [4].

authenticator_extension_output (optional)

Typ: Zusammengesetzt

Länge: 0 bis $(2^6 - 1) * (2^5 + 2^{16} + 1)$ Bytes + 1 Byte für die Anzahl der Elemente

Dieser Eintrag beinhaltet Antworten auf Erweiterungen durch den Authentifikator [4]. Aus ihm kann eine CBOR Map nach [6] mit der ID als Schlüssel und den Daten als Wert generiert werden. Es können maximal 63 Elemente übertragen werden.

extension_id

Typ: UTF-8 String

Länge: 0 bis $2^5 - 1$ Bytes + 1 Längen-Byte

Dieser Eintrag beinhaltet die eindeutige Kennung einer Erweiterung [4].

extension_data

Typ: JSON-codierbarer Wert. Von Erweiterung definiert.
Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes
Dieser Eintrag beinhaltet die Antwort auf eine Authentifikatorerweiterung [4].

signature

Typ: Bytestring
Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes

Dieses Feld beinhaltet die vom Authentifikator generiert Signatur [4].

user_handle (optional)

Typ: Bytestring
Länge: 0 bis $2^8 - 1$ Bytes + 1 Längen-Byte

Dieser Eintrag beinhaltet die Nutzer-ID, welche an die verwendeten PKCS gebunden ist [4].

selected_credential_id (optional)

Typ: Bytestring
Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes

Die „selected_credential_id“ beschreibt die vom Authentifikator ausgewählte PKCS. Wurde in „allow_credentials“ die Kennung genau einer PKCS übertragen, muss dieses Feld nicht verwendet werden [4].

client_extension_output (optional)

Typ: Zusammengesetzt
Länge: 0 bis $(2^6 - 1) * (2^5 + 2^{16} + 1)$ Bytes + 1 Byte für die Anzahl der Elemente

Dieser Eintrag beinhaltet Antworten auf Erweiterungen durch den Client [4]. Aus ihm kann eine CBOR Map nach [6] mit der ID als Schlüssel und den Daten als Wert generiert werden. Es können maximal 63 Elemente übertragen werden.

extension_id

Typ: UTF-8 String
Länge: 0 bis $2^5 - 1$ Bytes + 1 Längen-Byte
Dieser Eintrag beinhaltet die eindeutige Kennung einer Erweiterung [4].

extension_data

Typ: JSON-codierbarer Wert. Von Erweiterung definiert
Länge: 0 bis $2^{16} - 1$ Bytes + 2 Längen-Bytes
Dieser Eintrag beinhaltet die Antwort auf eine Clienterweiterung [4].

eph_user_name_client_share (optional)

Typ: Bytestring
Länge: 32 Bytes

In diesem Nachrichtenfeld werden 32 vom Client zufällig generierte Bytes übertragen, welche in die Generierung eines ephemeren Nutzernamens für eine spätere Verbindung eingehen.

Nachrichtenlänge

Die maximale Länge l, welche ein FIDO2AssertionResponse annehmen kann, berechnet sich wie unten angegeben. Eine TLS-Handshake-Nachricht darf maximal $m = 2^{24} - 1$ Bytes beinhalten [42]. Da gilt $l < m$, überschreitet keine FIDO2AssertionResponse die Maximallänge für Handshake-Nachrichten.

$$
\begin{array}{rr}
 & 1 \\
 & 2^{18} + 2 \\
 & 54 + 2 * (2^{16} + 1) + 63 * (2^5 + 2^{16} + 1) \\
 & 2^{16} + 1 \\
 & 2^8 \\
 & 2^{16} + 1 \\
 & 63 * (2^5 + 2^{16} + 1) + 1 \\
+ & 32 \\
\hline
 & 8.786.332 \text{ Bytes}
\end{array}
$$

Client-Nachricht: FIDO2AssertionResponse

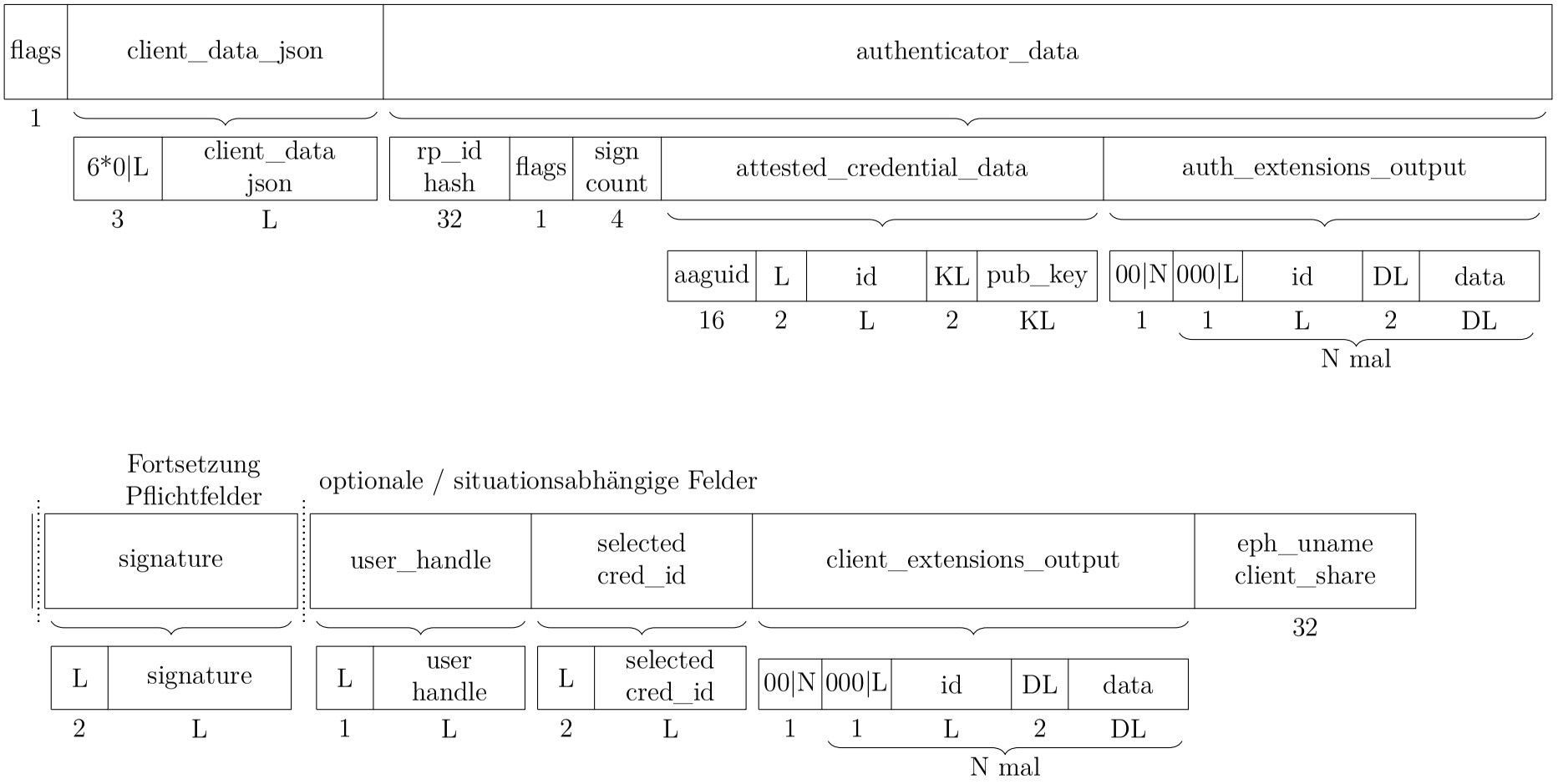

Abbildung B.4: Nachrichtenformat der FIDO2AssertionResponse
Quelle: Eigene Abbildung

Anhang C Alerts

C.1 fido2_bad_request

Kennung: fido2_bad_request = 151

Dieser Fehler signalisiert einem Endpunkt, dass die von ihm gestellte Anfrage des TFE-Handshakes fehlerhaft war.

C.2 fido2_authentication_error

Kennung: fido2_authentication_error = 152

Dieser Fehler signalisiert einem Client, dass die FIDO2-Authentifizierung wegen einer fehlerhaften Antwort fehlgeschlagen ist.

C.3 fido2_required

Kennung: fido2_required = 153

Dieser Fehler signalisiert einem Client, welcher in seiner ClientHello-Nachricht keine FIDO2-ClientHelloExtension übertragen hat, dass FIDO2 vom Server zwingend verlangt wird. Alternativ kann ein „insufficient_security“-Alert verwendet werden.

Anhang D Code Beispiele

D.1 Client

```
from socket import *
from tlslite.api import *

address = ("localhost", 4443)
sock = socket.socket(AF_INET, SOCK_STREAM)
sock.connect(address)

connection = TLSConnection(sock)
connection.handshakeClientFIDO2(address[0],
                                "erikamusterfrau")

connection.send(b"GET␣/␣HTTP/1.0\r\n\r\n")
while True:
    try:
        r = connection.recv(10240)
        if not r:
            break
    except socket.timeout:
        break
    except TLSAbruptCloseError:
        break
connection.close()
```

Abbildung D.1: HTTPS-Client mit Unterstützung für die FIDO2-Erweiterung
Quelle: Modifizierter Code auf Grundlage von [29]

D.2 Server

```
from socketserver import *
from http.server import *
from tlslite.api import *
import struct

cert_string = open("server_cert.pem", "rb").read()
cert_string = str(cert_string, 'utf-8')
cert_chain = X509CertChain()
cert_chain.parsePemList(cert_string)
key_string = open("server_key.pem", "rb").read()
key_string = str(key_string, 'utf-8')
private_key = parsePEMKey(key_string, private=True,
                          implementations=["python"])
address = ("localhost", 4443)
fido2_params = {'db_path': "fido2.db", 'rp_id': address[0]}

class MySimpleHTTPHandler(SimpleHTTPRequestHandler):
    wbufsize = -1
class MyHTTPServer(ThreadingMixIn, TLSSocketServerMixIn,
                   HTTPServer):
    def __init__(self, address, request_handler):
        HTTPServer.__init__(self, address, request_handler)

    def handshake(self, connection):
        connection.setsockopt(socket.IPPROTO_TCP,
                              socket.TCP_NODELAY, 1)
        connection.setsockopt(socket.SOL_SOCKET,
                              socket.SO_LINGER,
                              struct.pack('ii', 1, 5))
        connection.handshakeServer(certChain=cert_chain,
                                   privateKey=private_key,
                                   fido2_params=fido2_params)

        cc = connection.session.clientCertChain
        if cc and cc.is_fido2_cert_chain():
            print(cc.get_fido2_user_name() + \
                  "␣authenticated␣using␣FIDO2")
        return True

httpd = MyHTTPServer(address, MySimpleHTTPHandler)
httpd.serve_forever()
```

Abbildung D.2: HTTPS-Server mit Unterstützung für die FIDO2-Erweiterung
Quelle: Modifizierter Code auf Grundlage von [29]

```
CREATE TABLE users(
user_id BINARY(64) NOT NULL PRIMARY KEY,
user_name VARCHAR(50),
display_name VARCHAR(50)
);

CREATE TABLE certificates(
user_id BINARY(64) NOT NULL,
mode INTEGER NOT NULL,
certificate VARCHAR(65536) NOT NULL,
PRIMARY KEY (user_id, mode),
FOREIGN KEY (user_id) REFERENCES users(user_id)
);

CREATE TABLE credentials(
credential_id VARBINARY(65536) NOT NULL PRIMARY KEY,
aaguid BINARY(16) NOT NULL,
public_key VARBINARY(131072) NOT NULL,
signature_counter INTEGER NOT NULL DEFAULT 0,
user_id BINARY(64) NULL NULL,
mode INTEGER NOT NULL,
FOREIGN KEY (user_id) REFERENCES users(user_id)
);

CREATE TABLE eph_user_names(
eph_user_name BINARY(32) NOT NULL PRIMARY KEY,
user_id BINARY(64),
valid_through DATETIME NOT NULL,
FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Abbildung D.3: Datenbankschema der Nutzerdatenbank
Quelle: Eigene Abbildung

Literaturverzeichnis

[1] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley, Indianapolis (USA), 2001.

[2] D. Baghdasaryan, B. Hill, D. J. E. Hill, and D. Biggs, *FIDO Security Reference*, tech. rep., FIDO Alliance, 2018.

[3] D. Baghdasaryan, B. Hill, and J. Hodges, *FIDO Technical Glossary*, tech. rep., FIDO alliance, 2017.

[4] D. Balfanz, A. Czeskis, J. Hodges, J. Jones, M. B. Jones, A. Kumar, A. Liao, R. Lindemann, and E. Lundberg, *Web Authentication: An API for accessing Public Key Credentials*, tech. rep., W3C Recommendation, 2019.

[5] A. Beutelspacher, J. Schwenk, and K. Wolfenstetter, *Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge*, Springer, Wiesbaden (Germany), 8th ed., 2015.

[6] C. Bormann and P. E. Hoffman, *RFC 7049: Concise Binary Object Representation (CBOR)*, tech. rep., RFC Editor, 2013.

[7] S. Bradner, *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*, tech. rep., RFC Editor, 1997.

[8] C. Brand, A. Czeskis, J. Ehrensvärd, M. B. Jones, A. Kumar, R. Lindemann, A. Powers, and J. Verrept, *Client to Authenticator Protocol (CTAP)*, tech. rep., FIDO Alliance, 2018.

[9] F. Brecht, B. Fabian, S. Kunz, and S. Mueller, *Are you willing to wait longer for internet privacy?*, in European Conference on Information Systems, Helsinki (Finnland), 2011.

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

[11] J. Colnago, S. Devlin, M. Oates, C. Swoopes, L. Bauer, L. F. Cranor, and N. Christin, *It's not actually that horrible: Exploring Adoption of Two-Factor Authentication at a University*, in Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, Montreal (Canada), 2018.

[12] O. Dubuisson and P. Fouquart, *ASN.1: Communication Between Heterogeneous Systems*, Morgan Kaufmann, San Francisco (USA), 2000.

[13] D. E. Eastlake 3rd, *RFC 6066: Transport Layer Security (TLS) Extensions: Extension Definitions*, tech. rep., RFC Editor, 2011.

[14] C. Eckert, *IT-Sicherheit : Konzepte - Verfahren - Protokolle*, Oldenbourg, München (Germany) [u.a.], 8th ed., 2013.

[15] FIDO-Alliance, *Alliance Overview*. https://fidoalliance.org/overview. Zugriff am 22.07.2019.

[16] FIDO-Alliance, *FIDO Members*. https://fidoalliance.org/members. Zugriff am 22.07.2019.

[17] FIDO-Alliance, *FIDO2: WebAuthn & CTAP*. https://www.fidoalliance.org/fido2. Zugriff am 22.07.2019.

[18] FIDO-Alliance, *History of FIDO Alliance*. https://fidoalliance.org/overview/history. Zugriff am 22.07.2019.

[19] FIDO-Alliance, *How FIDO works*. https://fidoalliance.org/how-it-works. Zugriff am 22.07.2019.

[20] FIDO-Alliance, *Specifications Overview*. https://www.fidoalliance.org/specifications. Zugriff am 22.07.2019.

[21] Free Software Foundation, *GNU Lesser General Public License Version 2.1*. https://www.gnu.de/documents/lgpl-2.1.en.html, 1999. Zugriff am 06.08.2019.

[22] H. L. V. Gong, D. Balfanz, A. Czeskis, A. Birgisson, and J. Hodges, *FIDO 2.0: Web API for accessing FIDO 2.0 credentials*, tech. rep., FIDO Alliance dard, 2015.

[23] C. Herley, P. v. Oorschot, and A. S. Patrick, *Passwords: If We're So Smart, Why Are We Still Using Them?*, in Financial Cryptography and Data Security, R. Dingledine and P. Golle, eds., vol. 5628, Berlin, Heidelberg (Germany), 2009, Springer.

[24] I. Hickson, R. Berjon, S. Faulkner, et al., *HTML*, tech. rep., W3C Living Standard, 2019.

[25] D. Hühnlein, *Identitätsmanagement*, Datenschutz und Datensicherheit, 32 (2008), pp. 161–163.

[26] D. Hühnlein, C. Ziske, T. Hühnlein, T. Wich, D. Nemmert, S. Rohr, M. Hertlein, and C. Kölble, *Starke Authentisierung – jetzt!*, in D-A-CH Security, München (Deutschland), 2017.

[27] B. Ives, K. R. Walsh, and H. Schneider, *The Domino Effect of Password Reuse*, Commun. ACM, 47 (2004), pp. 75–78.

[28] C. Jacomme and S. Kremer, *An Extensive Formal Analysis of Multi-factor Authentication Protocols*, in IEEE 31st Computer Security Foundations Symposium (CSF), Los Alamitos (USA), 2018, IEEE Computer Society, pp. 1–15.

[29] H. Kairo, *tlslite-ng*. https://github.com/tomato42/tlslite-ng/tree/v0.8.0-alpha26. Version 0.8.0-alpha26.

[30] M. Khari, G. Shrivastava, S. Gupta, and R. Gupta, *Role of Cyber Security in Today's Scenario*, in Cyber Security and Threats, IGI Global, Hershey (USA), 2018, pp. 1–15.

[31] J. M. Kizza, *Guide to Computer Network Security*, Springer, London (UK) [u.a.], 4th ed., 2017.

[32] J. Kurose and K. Ross, *Computer Networking - A Top Down Approach*, Pearson, Boston (USA), 6th ed., 2013.

[33] B. Leiba, *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*, tech. rep., RFC Editor, 2017.

[34] Mozilla, *WebAuthn API Guide*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API. Zugriff am 22.07.2019.

[35] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, *The Cost of the "S" in HTTPS*, in Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14, Sydney (Australia), 2014, ACM, pp. 133–140.

[36] L. O'Gorman, *Comparing Passwords, Tokens, and Biometrics for User Authentication*, Proceedings of the IEEE, 91 (2003), pp. 2019–2020.

[37] H. S. Oluwatosin, *Client-Server Model*, IOSR Journal of Computer Engineering, 16 (2014), pp. 67–71.

[38] V. N. Padmanabhan and J. C. Mogul, *Improving HTTP latency*, Computer Networks and ISDN Systems, 28 (1995), pp. 25 – 35. Selected Papers from the Second World-Wide Web Conference.

[39] Ponemon Institute, *The 2019 State of Password and Authentication Security Behavior Report*. Research sponsored by Yubico, 2019.

[40] M. Reda, *Embedding of U2F into TLS 1.3*, Bachelor's Thesis, Humboldt-Universität zu Berlin, 2018.

[41] E. Rescorla, *SSL and TLS - Designing and Building Secure Systems*, Addison Wesley, Boston (USA), 2001.

[42] E. Rescorla, *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3*, tech. rep., RFC Editor, 2018.

[43] E. Rescorla and B. Korver, *RFC 3552: Guidelines for Writing RFC Text on Security Considerations*, tech. rep., RFC Editor, 2003.

[44] J. Reynolds, T. Smith, K. Reese, L. Dickinson, S. Ruoti, and K. Seamons, *A Tale of Two Studies: The Best and Worst of YubiKey Usability*, in 2018 IEEE Symposium on Security and Privacy (SP), Los Alamitos (USA), 2018, IEEE Computer Society, pp. 872–888.

[45] P. Saint-Andre, *RFC 8266: Preparation, Enforcement and Comparison of Internationalized Strings Representing Nicknames*, tech. rep., RFC Editor, 2017.

[46] J. A. Salowey and S. Turner, *RFC 8447: IANA Registry Updates for TLS and DTLS*, tech. rep., RFC Editor, 2018.

[47] J. Schaad, *RFC 8152: CBOR Object Signing and Encryption (COSE)*, tech. rep., RFC Editor, 2017.

[48] B. Schneier, *Applied cryptography : protocols, algorithms, and source code in C*, Wiley, New York (USA)[u.a.], 2nd ed., 1995.

[49] B. Schneier, *Secrets & Lies: Digital Security in a Networked World*, Wiley, New York (USA), 1st ed., 2000.

[50] S. Srinivas, D. Balfanz, E. Tiffany, and A. Czeskis, *Universal 2nd Factor (U2F) Overview*, tech. rep., FIDO Alliance Proposed Standard, 2017.

[51] S. Turner, *Transport Layer Security*, IEEE Internet Computing, 18(6) (2014), pp. 60–63.

[52] S. Wei, *Computer Network Security Technology Research*, in International Conference on Applications and Techniques in Cyber Security and Intelligence ATCI 2018, J. Abawajy, K.-K. R. Choo, R. Islaml, Z. Xu, and M. Atiquzzaman, eds., Shanghai (China), 2019, Springer International Publishing, pp. 824–831.

[53] Yubico, *python-fido2*. https://github.com/Yubico/python-fido2/tree/0.5.0. Version 0.5.0.

[54] Yubico, *WebAuthn Developer Guide - Overview*. https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/Overview.html. Zugriff am 20.08.2019.

[55] Yubico, *WebAuthn Developer Guide - Resident Keys*. https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/Resident_Keys.html. Zugriff am 20.08.2019.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig 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 24. Februar 2020 .....