Bachelor Thesis

W3C Web Cryptography API
Übersicht, Stand und Möglichkeiten

Martin Becker

Datum: 09. Oktober 2013
Betreuer: Vladislav Mladenov

Ruhr-Universität Bochum, Germany

Lehrstuhl für Netz- und Datensicherheit
Prof. Dr. Jörg Schwenk
Homepage: www.nds.rub.de

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für ein anderes Examen eingereichten Arbeit. Ich erkläre weiterhin, dass ich die Arbeit nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen und bildliche Darstellungen und dergleichen.

Ort, Datum	Unterschrift

Acknowledgements

Vladislav Mladenov, Florian Feldmann sowie meiner Familie.

Abstract

Bei dem *Web Cryptography API* Standard handelt es sich um eine Spezifikation für eine JavaScript-API, die es Webentwicklern ermöglichen soll, die kryptografischen Funktionalitäten des Browsers zu benutzen. Dazu wird von der Webanwendung JavaScript Code an den Client gesendet, wo die Operationen durchgeführt werden und der kryptografisch bearbeitete Inhalt an den Server zurück gesendet wird. Die Spezifikation ist noch nicht final, diese Arbeit bezieht sich auf den, zu diesem Zeitpunkt aktuellsten Stand, dies ist der *Editor's Draft 3 June 2013*, sowie auf die dazugehörigen Use Cases aus *Web Cryptography API Use Cases W3C Editor's Draft 02 June 2013*. Erweiterungen des Standards finden sich in dem Dokument *WebCrypto Key Discovery 8 January 2013*.
Ziel dieser Arbeit soll es sein, den aktuellen Stand der Entwicklung und gegenwärtige Probleme zu erläutern. Nach Möglichkeit sollen auch Lösungsvorschläge für einzelne offene Punkte bereitgestellt werden. Ebenso sollen die vorgesehenen Use Cases untersucht und ggf. anhand eines etablierten Protokolls für den entsprechenden Use Case verdeutlicht werden.
Da an diesem Standard derzeit noch gearbeitet wird ist es sehr wahrscheinlich, dass sich, während diese Thesis bearbeitet wird, einzelne Teile ändern, neue Probleme und offene Fragen entstehen, sowie bestehende Probleme gelöst werden. Diese Entwicklungen werden, nach Möglichkeit, in die Betrachtung aufgenommen. Dennoch besteht die Möglichkeit, das bei der Abgabe, einzelne Teile bereits obsolet sind.
Da es noch keine browserseitige Unterstützung für die API gibt, ist eine Implementierung nicht praktikabel und wird daher auch nicht durchgeführt.
Die Arbeit gliedert sich in 4 Bereiche:

1. Beschreibung und Erläuterung des Standards
2. Anwendungsfälle (Spezifiziert)
3. Anwendungfälle (Mögliche)
4. Stand der Entwicklung

KEYWORDS: Web Cryptography, Crypto, Java Script

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In modernen Browsern ist eine große Bandbreite an kryptografischen Funktionen integriert. Das Problem ist, dass es für Webentwickler bisher nicht möglich ist auf diese Funktionen zuzugreifen. Die *Web Cryptography Working Group* des *W3C* beschäftigt sich seit 2012 mit der Lösung dieses Problems. Das Ergebnis ist die Spezifikation für die *Web Cryptography API*, deren finale Fassung für das 2. Quartal 2014 erwartet wird. Mit Hilfe der Funktionen dieser API soll es möglich sein, mit JavaScript auf die kryptografischen Methoden des Browsers oder allgemeiner der User Agents, zuzugreifen. So soll eine Webanwendung einen Prozess anstoßen, der im User Agent z.B. ein Dokument signiert und das signierte Dokument an die Webanwendung übermittelt. Damit soll der User Agent Aufgaben übernehmen, für die zur Zeit noch spezielle Anwendungen benötigt werden.
Ziel dieser Arbeit ist eine Analyse des Standards und seiner Möglichkeiten. Dazu werden folgende Punkte betrachtet:

Analyse des Standards: Im Rahmen der Analyse des Standards wird der Aufbau der API betrachtet und die internen Abläufe erläutert. Ebenso werden alle Komponenten der API im Detail beschrieben.

Spezifizierte Anwendungfälle: Im Rahmen der Spezifikation werden einige Mögliche Anwendungsfälle von den Autoren vorschlagen. Diese Anwendungsfälle werden analysiert und aufgearbeitet, um ein besseres Verständnis zu ermöglichen.

Mögliche Anwendugsfälle: Neben den vorgeschlagenen Anwendungsfällen sollen weitere Anwendungsfälle gesucht und beschrieben werden, die mit Hilfe der Funktionalitäten der API umsetzbar sind.

Implementierungstand: Abschließend soll noch untersucht werden, in wie weit die Spezifikation bereits umgesetzt worden ist. Dazu wird zum einen analysiert, ob bereits Implementierungen in Browsern existieren und zum anderen, ob bereits Webanwendungen verfügbar sind, die die API einsetzen.

Da dieser Standard zum Zeitpunkt der Erstellung dieser Arbeit noch nicht final war, bezieht sich diese Arbeit im Schwerpunkt auf den *Web Cryptography API Editors Draft* vom 03.06.2013, die *Web Cryptography API Use Cases Editors Draft* vom 02.06.2013 sowie auf den *WebCrypto Key Discovery Working Draft* vom 08.01.2013. An wenigen Stellen bezieht sich die Arbeit auf eine aktuellere Version der genannten Dokumente, dies wird jedoch im entsprechenden Abschnitt angegeben. Aus diesem Grund ist es auch möglich, dass Probleme, die in dieser Arbeit angesprochen werden, bereits gelöst wurden. Ebenso ist es möglich, dass neue Probleme aufgetreten sind, die nicht betrachtet werden. Es wurde zwar darauf geachtet, dass wichtige Änderungen noch berücksichtigt wurden, allerdings besteht in diesem Fall kein Anspruch auf Vollständigkeit.

Es sei an dieser Stelle auch noch darauf hingewiesen, dass im Rahmen dieser Arbeit keine Implementierungen durchgeführt werden, ebenso findet keine Betrachtung der serverseitigen Vorgänge statt.
Zum Nachvollziehen der Anwendungsfälle ist ein gewisses Maß an kryptografischen Grundwissen notwendig. Diese Grundlagen werden in 2 kurz erläutert.

2. Grundlagen

Im Rahmen dieses Dokumentes werden keine tiefgehenden Kenntnisse der Kryptografie vorausgesetzt. Um die in Kapitel 4 und 5 beschriebenen Anwendungsfälle nachvollziehen zu können, werden die benötigten kryptografischen Grundlagen und Schreibweisen, die verwendet werden, an dieser Stelle kurz vermittelt.

2.1. Schreibweisen

Der Index i repräsentiert die einzelnen Parteien

Schreibweise	**Bedeutung**
$priv_i$	Privater Teil des Schlüsselpaares von Partei i
pub_i	Öffentlicher Teil des Schlüsselpaares von Partei i
K	Geheimer, symmetrischer Schlüssel, den alle Parteien teilen
$enc_K(...)$	Verschlüsselung mit dem symmetrischen Schlüssel K
$enc_{pub_i}(...)$	Verschlüsselung mit dem öffentlichen Schlüssel von Partei i
$sig_{priv_i(...)}$	Signatur mit dem privaten Schlüssel von Partei i
N	Eine beliebige Nachricht
N_i	Eine Zufallszahl, die von Partei i erzeugt wird
U	Benutzerinformationen
`bsp`	Interface-, Methoden- und Typenbezeichnungen
`Typ` **`objekt`**	Typenbezeichnung mit instanziiertem Objekt. Die Objektinstanz ist zusätzlich **fett** gedruckt
`Methode()`	() hinter einer Bezeichnung weisen auf eine Methode hin

2.2. Kryptographie

In diesem Abschnitt wird kurz auf die Funktionsweise von symmetrischer- und asymmetrischer Kryptografie eingegangen. Vertiefende Informationen können [1] entnommen werden.

2.2.1. Hash

Eine Hash-Funktion bildet eine beliebig lange Bytesequenz auf eine Bytesequenz fester Länge ab. Heute übliche Längen sind 160 bis 512 Bit. Dabei erzeugt jede Eingangssequenz eine andere Ausgangssequenz, wobei eine kleine Änderung im Eingang eine große Änderung im Ausgang bewirkt. Hash-Funktionen sind Einwegfunktionen. Das bedeutet, dass es einfach ist, aus einer gegebenen Eingangsfolge eine Ausgangs-

folge zu berechnen. Aus einer gegebenen Ausgangsfolge die Eingangsfolge zu berechnen ist dagegen sehr aufwendig.

2.2.2. Asymmetrische Kryptografie

Die asymmetrische Kryptografie verwendet Schlüsselpaare, bestehend aus einem privaten und einem öffentlichen Teil. Der öffentliche Teil wird veröffentlicht, dabei muss nicht auf sichere Wege geachtet werden. Wenn der öffentliche und der private Schlüssel zusammen auf eine beliebige Nachricht N angewendet werden ergibt sich wieder N. Die Reihenfolge der Anwendung spielt dabei keine Rolle.
Wenn eine Partei einer Anderen Daten vertraulich übermitteln möchte, verschlüsselt der Absender die Daten mit dem öffentlichen Schlüssel des Empfängers. Wenn dieser nun seinen privaten Schlüssel auf das Chiffrat anwendet, erhält er die übermittelten Daten im Klartext.
Umgekehrt verhält es sich mit digitalen Signaturen. Von einer zu signierenden Nachricht wird zunächst der Hash-Wert gebildet, dieser wird anschließend mit dem privaten Schlüssel des Absenders verschlüsselt und zusammen mit der Nachricht im Klartext übermittelt. Der Empfänger wendet den öffentlichen Schlüssel des Absenders auf die Signatur an und erhält nun den Hash-Wert im Klartext. Anschließend wird der Hash-Wert der Nachricht vom Empfänger erzeugt und mit dem aus der Signatur verglichen. So können Manipulationen aufgedeckt werden. Im Gegensatz zum *Message Authetication Codes* (MACs), die im Abschnitt 2.2.3 erläutert werden, kann bei digitalen Signatur eindeutig festgestellt werden, wer die Nachricht signiert hat, da nur eine Partei Zugriff auf den privaten Schlüssel hat.

2.2.3. Symmetrische Kryptografie

Bei der symmetrischen Kryptografie teilen sich alle Parteien einen gemeinsamen geheimen Schlüssel. Die Verschlüsselung und die Entschlüsselung verwenden beide den selben Schlüssel K. Um eine sichere Kommunikation zu gewährleisten, muss dieser Schlüssel K zunächst sicher transportiert werden. Im Gegensatz zu asymmetrischen Algorithmen sind symmetrische Algorithmen sehr effizient zu implementieren, daher ist die Erzeugung eines Chiffrates sehr performant.
Als äquivalent zu den digitalen Signaturen der asymmetrischen Kryptografie kann man bei der symmetrischen Kryptografie *Message Authetication Codes* (MAC) verwenden. Dazu wird der Hash-Wert der Nachricht mit dem Schlüssel K verschlüsselt. Beim Empfänger wird die Prozedur wiederholt und die beiden Ergebnisse werden verglichen. Jeder, der Zugriff auf den Schlüssel hat, kann sowohl den MAC erzeugen als auch überprüfen. Es kann also nicht eindeutig festgestellt werden wer den MAC erzeugt hat.

3. Der Standard

Um ein Verständnis für die Anwendungsmöglichkeiten der *Web Cryptography API*, im folgenden *API* oder *WebCrypto API* genannt, zu bekommen, muss zunächst einmal ein Verständnis für die Zielsetzung und die Funktionsweise geschaffen werden. Aus diesem Grunde wird im folgenden Kapitel der Standard beleuchtet. Dazu werden folgende Fragen geklärt:

- Was sind die Ziele, was sind keine Ziele
- Welche Annahmen wurden getroffen
- Welche Sicherheitsaspekte müssen beachtet werden
- Funktionsweise der API
- Welche Komponenten besitzt die API
- Welche Funktionen und Algorithmen werden von der API unterstützt

3.1. Allgemeines

Durch die API sollen Webentwickler die Möglichkeit bekommen, auf kryptografische Funktionen innerhalb des User Agent zuzugreifen. User Agent bezeichnet dabei eine Anwendung, die in der Lage ist Webanwendungen aufzurufen, z.B. ein Browser. Dieser Zugriff soll durch JavaScript Methoden ermöglicht werden, die ihrerseits die im User Agent hinterlegten Crypto-Algorithmen aufrufen und das Ergebnis an JavaScript zurückgeben. Bisher war dies nicht möglich.
Die Implementierung der Crypto-Methoden soll dabei direkt im User Agent stattfinden und nicht als eine Erweiterung einer bestehenden Anwendung in Form eines PlugIn.
Inhalt dieser Spezifikation ist es festzulegen, welche Methoden und Objekte der Browser zur Verfügung stellen muss und wie sich diese verhalten müssen. Es wird keine Empfehlung für die Implementierung der Crypto-Algorithmen gegeben. Ebenso schreibt der Standard auch keine Crypto-Algorithmen vor, die Autoren empfehlen jedoch eine Auswahl an Algorithmen, um eine gewisse Interoperabilität zu erreichen. Weiter existiert keine Spezifikation für den Schlüsselspeicher, dieser liegt vollständig in der Verantwortung der User Agent Entwickler.

3.1.1. Ziele

Die Ziele der API sind in der Spezifikation [2] festgelegt, dennoch wurden diese zum Einen in der offiziellen Mailingliste [3] und zum Anderen von Harry Halpin auf dem *29. Chaos Communication Congress 2012* [4] ergänzt und näher beschrieben. Die folgenden Punkte setzten sich aus Aussagen aus dem Standard [2, Abschn. 4], der Mailingliste [3] und dem Vortrag [4] zusammen. Als Ziele werden genannt:

- Bereitstellen von Möglichkeiten, um Webanwendungen sicherer zu machen. Darunter fällt u.A. die Möglichkeit, sich mit Zertifikaten an einer Webanwendung zu authentifizieren oder eine End-to-End Verschlüsselung verwenden zu können. Mit Hilfe einer End-to-End Verschlüsselung wird sichergestellt, dass Daten nur beim Absender und beim Empfänger im Klartext vorliegen. Auf dem kompletten Transportweg sind die Daten verschlüsselt.
- Es soll die Möglichkeit geschaffen werden, gute und sichere Zufallszahlen zu erzeugen.
- Alle grundlegenden kryptografischen Operationen sollen verfügbar sein. Darunter fallen z.B. Verschlüsselung, Entschlüsselung und digitale Signaturen. Eine vollständige Auflistung aller Funktionen, die unterstützt werden sollen, gibt Tabelle 3.6.
- Vor allem durch langlebige Schlüssel kann es möglich sein, dass der Benutzer eindeutig identifiziert werden kann. Dies soll nach Möglichkeiten verhindert werden.

Ebenso wurden folgende Anwendungen explizit ausgeschlossen:

- Es wird explizit darauf hingewiesen, dass durch die Verwendung der API Webanwendungen nicht sicher sondern nur sicherer werden.
- Grundlegende Sicherheitsprobleme werden nicht gelöst. Darunter zählen u.A. unverschlüsselte Verbindungen, nicht standardkonformes Verhalten, das Verhalten und die Sicherheit der User Agents, sowie gebrochene kryptografische Verfahren.
- Durch diesen Standard sollen nicht die Probleme, die bei der Verwendung von JavaScript auftreten, behoben werden. Darunter fallen z.B. die Authentizität des JavaScript-Codes genau so, wie die allgemeinen Sicherheitsprobleme von JavaScript wie sie u.A. vom Bundesamt für Sicherheit in der Informationstechnik in [5] beschrieben werden.
- Die API ist kein Teil des Client-Server-Sicherheitsmodells. Im Speziellen soll dadurch nicht TLS/SSL ersetzt werden. Der Einsatz von TLS/SSL wird für alle sicherheitskritischen Anwendungen explizit empfohlen.
- Die API bietet keinen Schutz gegen *Cross-Site-Scripting* (XSS) Angriffe, bei denen eine Sicherheitslücke ausgenutzt wird, wobei Informationen aus einem Kontext, in dem sie nicht vertrauenswürdig sind, in einen anderen Kontext eingefügt werden, in dem sie als vertrauenswürdig eingestuft werden. Weiter bietet sie auch keinen Schutz *Cross-Site Request Forgery* (CSRF)-Attacken, ein Angriff,

bei dem Anwender ungewollt Aktionen in einer Webanwendung durchführen, bei der sie zu diesem Zeitpunkt, angemeldet sind.

- Die API selber implementiert weder kryptografische Methoden, noch erzeugt sie Schlüssel. Diese Funktionen werden nur durch API-Methoden angestoßen, aber vom User Agent ausgeführt.

Abschließend sei noch darauf hingewiesen, dass die API nur Operationen spezifiziert, die im User Agent auf dem Client stattfinden. Die Serverseite wird nicht betrachtet.

3.1.2. Annahmen und Voraussetzungen

Die Autoren des Standards definieren diverse Sicherheitsannahmen und Voraussetzungen, unter denen die API wie spezifiziert arbeitet. Die Rahmenbedingungen dafür schaffen andere Konzepte und Spezifikationen, auf die an dieser Stelle nicht weiter eingegangen wird. Wenn spezielle Konzepte für das Verständnis der Arbeitsweise benötigt werden, werden diese an entsprechender Stelle erläutert. Das Umfeld für die API ist in [2, 4] wie folgt definiert:

- Es wird davon ausgegangen, dass Server sowie Webanwendung sicher sind und das die Verbindung zwischen User Agent und Server via TLS/SSL gesichert ist.
- Die benötigten kryptografischen Methoden sind innerhalb des Browsers implementiert und nicht in einem PlugIn. Im Speziellen soll von Außen kein Einfluss auf die Verarbeitung der Daten innerhalb der kryptografischen Funktionen möglich sein.
- Nicht direkt als Voraussetzung formulierte Harry Halpin in [4] die Aussage:“ Wenn man annimmt, dass alles böse ist, bringt einem die API nichts“.

3.2. Sicherheitsüberlegungen

Diese Sicherheitsüberlegungen sind nicht explizit Teil des Standards, sondern sollen lediglich als Hinweise für Entwickler dienen. Dabei werden die Entwickler in zwei Gruppen geteilt. Zum einen existieren die *Implementors*, welche für die Entwicklung des User Agents verantwortlich sind, auf der anderen Seite sind die *Developer* genannt, die später ihre Webanwendungen mit dem entsprechenden JavaScript Code ausrüsten. Um die Unterscheidung der beiden Gruppen zu erleichtern, werden im Rahmen dieser Arbeit folgende Konventionen verwendet:

Implementors : User Agent Entwickler

Developers : Webentwickler

3.2.1. Sicherheitsüberlegungen für Implementors

User Agent Entwickler müssen sich über das Problem des Key Reuse, also das mehrfache Verwenden eines Schlüssels für verschiedene Webanwendungen, Gedanken machen. Die Spezifikation sieht diese Möglichkeit in [2, Abschn. 5.1] vor und bietet zwei Lösungsvorschläge, die allerdings problembehaftet sind.

1. Der User wird darauf hingewiesen, dass er einen Schlüssel mehrfach verwendet und wird auf die möglichen Sicherheitsrisiken hingewiesen. Ob es nun zu einer mehrfachen Schlüsselverwendung kommt, entscheidet der User selbst. Problematisch an einem solchen Vorgehen ist allerdings das, ähnlich wie bei TLS/SSL, der User eine Sicherheitsentscheidung treffen muss, ohne das er abschätzen kann welche Folgen diese Entscheidung hat. Dies wurde in [6] anhand des Benutzerverhaltens bei TLS/SSL Zertifikatsfehlermeldungen untersucht. Eine mögliche Folge ist es, dass eine Webanwendung die Kommunikation zu einer anderen entschlüsseln kann, da der User Agent den selben Schlüssel mit mehreren Parteien teilt.

2. Es existieren vordefinierte Vertrauensverhältnisse, bei denen ein Key Reuse automatisch und für den User transparent passiert. Dabei ist nicht spezifiziert, wie ein solches Vertrauensverhältnis aussehen könnte. Denkbar wäre aber eine Entscheidung anhand von Zertifikaten.

In [2, Abschn. 4.4] wird darauf hingewiesen, dass das Schlüsselmanagement ein Problem ist. So muss der Entwickler u.A. darauf achten, dass Schlüssel nur vom User Agent aus dem Schlüsselspeicher ausgelesen werden können.
Da keine Crypto-Algorithmen vorgeschriebenen sind, ist es ebenfalls möglich, eigene Algorithmen zu entwickeln, zu implementieren und zu registrieren.
Als Empfehlung für die Bereitstellung sicherer und guter Zufallszahlen wird, unter Linux, in [2, Abschn. 9.1], die Funktion */dev/random* bzw. */dev/urandom* genannt.

3.2.2. Sicherheitsüberlegungen für Developer

Webentwickler sollten sicher stellen, dass die in Abschnitt 3.1.2 beschriebenen Voraussetzungen erfüllt sind. Nach [2, Abschn. 5.2] sollten sich Webentwickler im klaren darüber sein, dass die Verwendung sicherer Kryptografie nicht automatisch bedeutet, dass der Anwendungsfall, in dem sie verwendet wird, sicher ist. Viele Beispiele dazu sind in [7], anhand von Authentifikation und Key Establishment Protokollen gegeben.

3.3. Privacy Betrachtungen

Durch die Verwendung von Schlüsseln, besonders bei Langzeitschlüsseln, besteht die Möglichkeit das Profile über Nutzer erstellt werden bzw. Profile Benutzern eindeutig zuzuordnen sind. Dies kann durch drei Mechanismen erreicht werden, die in [2, Abschn. 6], wie folgt beschrieben werden:

3.3.1. Fingerprinting

Durch die Kombination von verschieden Faktoren kann es möglich sein, dass Benutzer eindeutig zu identifizieren sind. Techniken zu dem Fingerprinting sind zwar auch ohne die Verwendung der API möglich, allerdings können Informationen wie Browser Version, Betriebssystem Version, Benutzername und IP-Adresse nun durch weitere Informationen ergänzt werden. Darunter fällt z.B., ob ein User Agent einen bestimmten Algorithmus unterstützt und ob er einen speziellen öffentlichen Schlüssel besitzt.
Die Gefahr, dass die unterstützen Algorithmen zur Identifizierung herangezogen werden, begegnet man laut [3], indem bewusst die Möglichkeit der Auflistung der Algorithmen nicht unterstützt wird.
Die Auflistung der öffentlichen Schlüssel sollte dahingehend unterbunden werden, dass nur autorisierte Webanwendungen auf den jeweiligen Schlüssel zugreifen können. Dies wird dadurch erreicht, dass die Webanwendung den Schlüssel, den der User Agent benutzen soll, angeben muss. Im Allgemeinen sollte für jede Webanwendung ein eigenes Schlüsselpaar erzeugt werden, dies ist auch im Sinne der Vermeidung von Key Reuse sinnvoll.

3.3.2. Tracking

Wenn ein Schlüssel für die Kommunikation mit mehreren Webanwendungen verwendet wird, ist es durchaus möglich, ein Profil über das Nutzungsverhalten des Benutzers zu erstellen. Dies setzt allerdings einen Austausch von Informationen zwischen verschiedenen Webanwendungen bzw. deren Betreibern voraus. Dies zeigt ebenfalls, dass es sinnvoll ist, für jede Webanwendung einen eigenen Schlüssel zu erzeugen. Dieses Thema wird im Abschnitt 3.4.2.3.5 noch einmal näher betrachtet.

3.3.3. Super Cookies

Aus der Tatsache, dass Schlüssel häufig eine sehr lange Lebensdauer haben, ist es ebenfalls möglich, diese als Cookies zu verwenden. So wird eine eindeutige Zuordnung auch über verschiedene User Agents möglich, solange die verwendeten Schlüssel auf allen User Agents gleich sind, was z.B. für ein Authentifizierung-Szenario denkbar ist. Eine Zuordnung ist so auch möglich, wenn herkömmliche Cookies regelmäßig gelöscht oder nicht angelegt werden.
Die einzige Möglichkeit, dies zu verhindern, wäre ein regelmäßiger Austausch der Schlüssel, was allerdings auch nicht zweckmäßig ist, wenn die Webanwendung in der Lage ist den alten und neuen Schlüssel in einen gemeinsamen Kontext zu bringen. Dies ist vorstellbar, da wie in [3] beschrieben, eine Webanwendung den Nutzer zuerst identifizieren muss, bevor sie entscheidet, welcher Schlüssel verwendet werden soll. So können Nutzer einem Schlüssel eindeutig zugeordnet werden. Auf diese Art könnten auch, von der Webanwendung gespeicherte, alte Schlüssel mit dem neuen Schlüssel verbunden werden.

3.4. Funktionsweise

Im folgenden Abschnitt wird die Funktionsweise der API beschrieben. Die Beschreibung orientiert sich dabei am Top-Down-Ansatz. Dazu wird erst ein allgemeiner Überblick über die Funktionsweise der gesamten API gegeben, bevor die Komponenten einzeln erläutert werden.

3.4.1. Schematische Darstellung

In Abb. 3.1 ist eine Übersicht gegeben, wie die Daten in einem Szenario fließen, in dem der Server dem User Agent aufträgt, eine einfache kryptografische Operation durchzuführen. Dieses Beispiel arbeitet mit der Voraussetzung, dass der gewünschte Algorithmus implementiert ist und der angeforderte Schlüssel im User-Agent hinterlegt ist.
Dazu sind folgende Schritte notwendig:

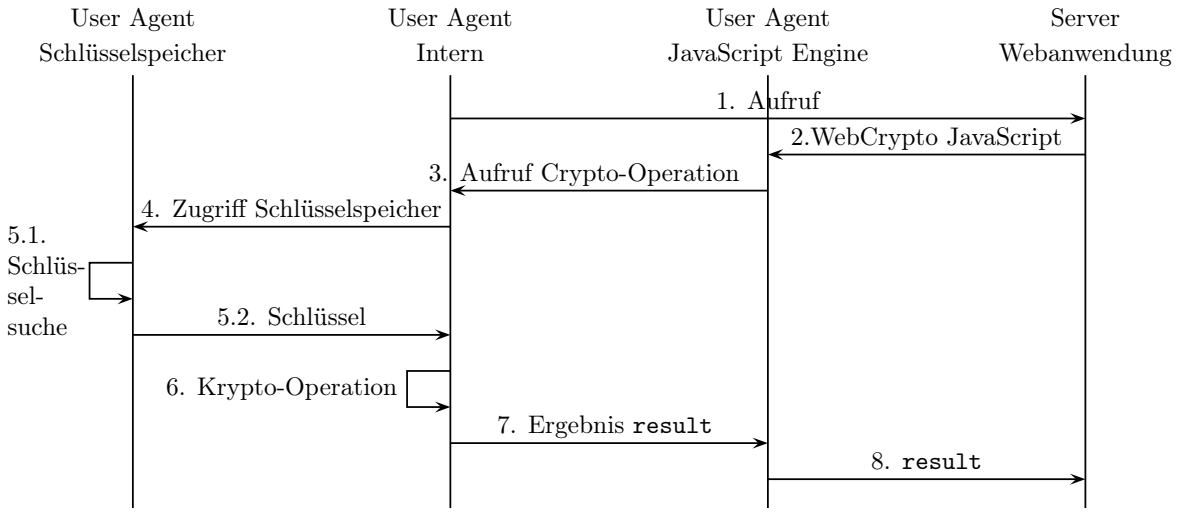


Abbildung 3.1.: Übersicht, [2]

1. Der User ruft mit Hilfe des User Agents eine Webanwendung auf.
2. Die Webanwendung sendet dem User Agent die angeforderte Ressource. Diese Ressource enthält den JavaScript Code, der den JavaScript-seitigen Teil der Web Crypto API implementiert.
3. Die JavaScript Engine des User Agents interpretiert den empfangenen JavaScript Code und ruft die angeforderte Crypto-Operation auf. Dafür übergibt der JavaScript Code die notwendigen Parameter:
   - Daten, auf denen die Crypto-Operation ausgeführt werden soll.
   - Den Algorithmus der angewendet werden soll.

- Die notwendigen Informationen, um den Schlüssel zu lokalisieren (Optional für den Fall, dass nur ein Hash-Wert gebildet werden soll).

4. Der aufgerufene Crypto-Algorithmus greift auf den Schlüsselspeicher zu (dies entfällt bei der Erzeugung eines Hash-Wertes mit der `Digest()`-Methode).
5. Der Schlüsselspeicher sucht den Schlüssel und gibt ihn an die aufrufende Crypto-Operation zurück.
6. Mit dem Schlüssel wird nun die eigentliche Crypto-Operation im User Agent durchgeführt.
7. Das Ergebnis der Operation wird an die JavaScript Engine zurückgegeben.
8. Die JavaScript Engine gibt das Ergebnis der Crypto-Operation an die Webanwendung zurück.

Wichtig zu erwähnen ist an dieser Stelle, dass die JavaScript Engine zu keiner Zeit einen Schlüssel im Klartext sehen darf und alle Crypto-Operationen im User Agent selber stattfinden.

3.4.2. Komponenten der API

Im folgenden Abschnitt werden die einzelnen Komponenten der API genauer betrachtet und deren Funktionsweise schematisch erläutert. Dies soll zu einem besseren Verständnis der inneren Abläufe der API führen.

3.4.2.1. Algorithm Dictionary

Das *Algorithm Dictionary* wird in [2, Abschn. 10] eingeführt und ist ein Dictionary Objekt, das einem Algorithmus bestimmte dazugehörige Operationen zuordnet. Als Identifier wird ein Name im Form eines `DOMStrings` benötigt. Zum Beispiel ordnet der Identifier `AES-CTR` diesem Algorithmus die Operationen *encrypt(), decrypt()* und *generateKey()* zu, es kann also keine Signatur erstellt werden. Das gleiche gilt für `SHA-1`, der wiederum nur die Operation *digest()* unterstützt. Eine Übersicht über mögliche Algorithmus/-Operationszuordnungen gibt [2, Abschn. 17.1] sowie Tab. 3.10.

3.4.2.2. RandomSource-Interface

Das `RandomSource-Interface`, wird in [2, Abschn. 9] eingeführt und stellt eine Schnittstelle für die Bereitstellung von Zufallswerten da. Die Implementierung ist dem User Agent Entwickler überlassen, dennoch wird empfohlen, dass die Entropie aus einer zuverlässigen Quelle wie `/dev/random` bzw. `/dev/urandom` stammt. Laut [2, Abschn. 9.1] ist allerdings kein Minimum an Entropie spezifiziert.
Der Zugriff auf die Zufallswerte erfolgt über die `getRandomValue()` Funktion. Diese ist laut [2, Abschn. 9.2.1], nicht für die Erzeugung von Schlüsseln vorgesehen. Dafür ist die Funktion `generateKey()` zu verwenden, da diese die Besonderheiten der einzelnen Algorithmen berücksichtigt.

Der in Abb. 3.2 gezeigte und unten beschriebene Ablauf beschreibt das spezifizierte Verhalten des User Agents bei dem Aufruf der JavaScript Methode `getRandomValue()`:

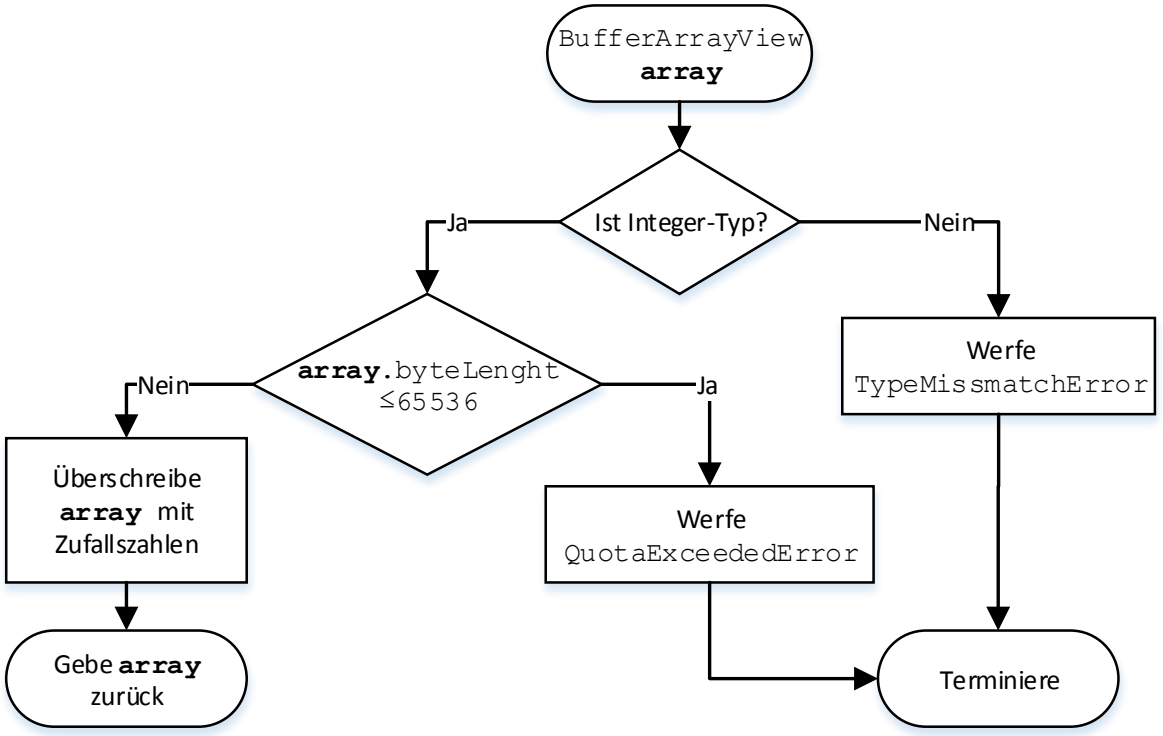


Abbildung 3.2.: getRandomValue(), [2, Abschn. 9]

Eingabe: `ArrayBufferView` **`array`**
Ausgabe: `ArrayBufferView`

- Als Eingabe wird ein `ArrayBufferView` **`array`** übernommen
- Es wird geprüft, ob **`array`** ein Integer-Typen hat. Wenn nicht, wird einen `TypeMissmatchError` geworfen und der Algorithmus terminiert. Sonst wird der Algorithmus fortgesetzt.
- Wenn **`array`** ein Interger-Typen, wird überprüft, ob es eine Maximale Länge von 65536 Byte hat.
- Wenn `array` die erlaubte Länge nicht überschreitet, wird `array` mit Zufallswerten überschrieben und `array` als Ergebnis zurückgegeben. Andernfalls wird ein `QuotaExeededError` geworfen und der Algorithmus terminiert.

3.4.2.3. Schlüssel

Der folgende Abschnitt beschäftigt sich mit dem Schlüsselhandling innerhalb der API und nennt Punkte, die bei der Implementierung und Verwendung zu beachten sind.

3.4.2.3.1. Key-Interface Das in [2, Abschn. 11] eingeführte `Key-Interface` ist eines der zentralen Interfaces der API und ermöglicht eine eindeutige Referenz auf einen Schlüssel, der vom User Agent verwaltet wird. Das Interface stellt zum einen Enumerations bereit, welche in Tab.3.1 aufgelistet sind, sowie die eigentlichen Member, welche in Tab.3.2 erläutert sind.

Enumeration	**Werte**	**Beschreibung**
`Enumeration` **`KeyType`**	secret public private	Spezifiziert den Typ des Schlüssels
`Enumeration` **`KeyUsage`**	encrypt decrypt sign verify deriveKey wrap unwrap deriveBits	Spezifiziert die von dem Schlüssel unterstützen Operationen (Erläuerung der Methoden in Tab. 3.6)

Tabelle 3.1.: Enumerations Key-Interface, [2, Abschn. 11]

Attribut	**Beschreibung**
`bool` **`extractable`**	Gibt an, ob ein Schlüssel mit Hilfe der Methode `wrap()` extrahiert werden kann
`Algorithm` **`algorithm`**	Beschreibt den zu dem Schlüssel gehörenden Algorithmus
`KeyType` **`type`**	Kann einen Wert aus dem `Enumeration` **`KeyType`** annehmen
`KeyUsage[]` **`keyUsage`**	Array, welches angibt, für welche Operation aus der `Enumeration` **`KeyUsage`** der Schlüssel verwendet werden kann

Tabelle 3.2.: Member Key-Interface, [2, Abschn. 11.2]

3.4.2.3.2. Structured Clone Algorithm In [2, Abschn. 11.3] wir die Möglichkeit spezifiziert, einen Schlüssel zu klonen. Dazu muss ein neues Key-Objekt erzeugt werden, das alle Attribute des Quell-Objekts übernimmt. Dabei darf das Schlüsselmaterial niemals an JavaScript übergeben werden, was dazu führt, dass der geklonte Schlüssel nur innerhalb des User Agents verfügbar ist und ggf. als eigenes Objekt für die Kommunikation mit einer anderen Webanwendung verwendet werden kann.

3.4.2.3.3. KeyDiscovery Das `NamedKey-Interface` dient der Suche nach einem passenden Schlüssel für eine Webanwendung und wird in [8, Abschn. 7.2] eingeführt. Dazu wird das `Key-Interface` um die Attribute `DomString` **`Name`** und `DomString` **`GUID`** erweitert, wobei die `GUID` ein eindeutiger, Base64 codierter Identifier ist, der mindestens 256 Bit Länge hat. Der `Name` ist ein lokaler Identifier für den Schlüssel. Wie genau die Webanwendung die passende `GUID` und den passenden `Name` für eine Verbindung ermittelt, ist dem Webentwickler überlassen und wird nicht beschrieben.
Neben dem `NamedKey-Interface` gehört auch das `CryptoKeys-Interface` zu dem *Key Discovery* Konstrukt. Diese Interface wird in [8, Abschn. 7.3] eingeführt und stellt lediglich die Methode `getKeysByName()` mit dem Eingabewert `DomString` **`Name`** bereit. Der `DomString` **`Name`** entspricht dabei dem aus dem `NamedKey-Interface`.
Die Schlüsselsuche, welche in Abb. 3.3 grafisch dargestellt ist, ist ein asynchroner Vorgang, der das *future-Pattern*, ein Entwurfsmuster für asynchrone Verarbeitung, welches in A.1.1 beschrieben wird, implementiert. Um einen Schlüssel zu finden, muss der User Agent folgende Schritte durchführen:

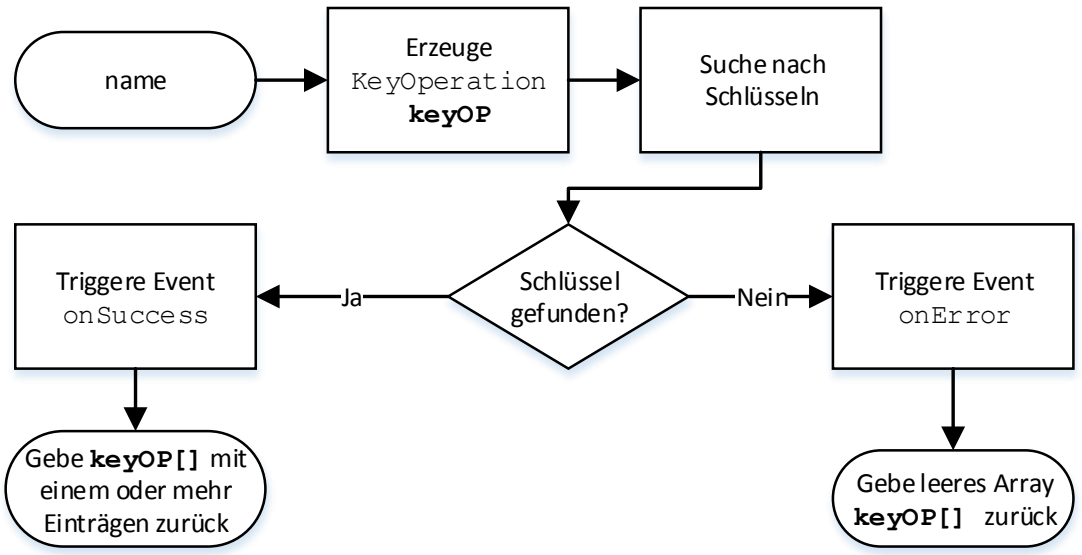


Abbildung 3.3.: GetKeysByName(), [8, Abschn. 7.3.1]

Eingabe: `DOMString` **`name`**
Ausgabe: `KeyOperation[]` **`keyOP[]`**

- Als Eingabewert wird der `DOMString` **`name`** an die Funktion übergeben.
- Erstellen eines neuen Objektes **`keyOp`** vom Typ `KeyOperation`. Das `KeyOperation-Interface` ist weder im Standard [2] noch im Ergänzungsdokument *Key Discovery* [8] beschrieben.
- Die eigentliche Suche des Schlüssels wird durchgeführt. Da dies von der Implementierung des Schlüs-

selspeichers abhängig ist, liegt die Verantwortung für die Funktionsweise der Suche bei dem User Agent Entwickler. Der Schlüssel gilt als gefunden, wenn die *Unicode* Repräsentation eines Schlüssels im Speicher mit der *Unicode* Repräsentation des Eingabewertes übereinstimmt. *Unicode* ist ein Standard, in dem die Codierung von Schriftzeichen festgelegt ist, so dass „A“ von allen User Agents als „A“ erkannt wird.

- Wenn ein oder mehrere Schlüssel gefunden wurden, muss das Ereignis `onSuccess()` getriggert werden, welches der asynchronen Operation mitteilt, das der Prozess abgeschlossen ist und ein Ergebnis vorliegt. Wenn kein Schlüssel gefunden wurde, muss das `onError()` Ereignis getriggert werden, welches den asynchronen Prozess beendet und ein leeres Array zurück gibt. Wichtig hierbei ist, dass ein leeres Array und nicht `null` zurück gegeben wird, da der Wert `null` bei der Übergabe in eine Crypto-Operation bedeutet, dass kein Schlüssel benötigt wird.

Um das *Key Discovery* verwenden zu können, müssen das `Window-Interface`, welches in [9] beschrieben wird und eine Schnittstelle zum aktuellen Browserfenster bereitstellt, sowie das `WorkerGlobalScope`, das in [10] spezifiziert ist und Zugriff auf im Hintergrund laufende Scripte ermöglicht, um das Attribute `CryptoKeys` erweitert werden. Diese Erweiterungen werden in [8, Abschn. 7.4, 7.5] spezifiziert.

3.4.2.3.4. Name des Schlüssels In der Spezifikation ist nicht festgelegt, wie ein Schlüssel benannt werden muss oder wie eine Webanwendung den Namen des anzufordernden Schlüssels erfährt. Auszuschließen ist an dieser Stelle aber die Möglichkeit, dass der User Agent der Webanwendung den Schlüssel mitteilt, da keine entsprechende Methode spezifiziert wird. Weitere Möglichkeiten wären die Verwendung von etablierten Erkennungstechniken wie *Username / Passwort* oder auch Cookies. Nach Aussage von Mark Watson[1] [3] ist es vorgesehen, dass die Webanwendung den Namen des Schlüssels kennen muss. Das Szenario, für das der *Key Discovery* gedacht ist, wird von Mark Watson in [3] beschrieben und stellt sich wie in Abb. 3.4 da. Dazu müssen folgende Voraussetzungen erfüllt werden:

- Vom Hersteller eines Gerätes, hier wird speziell der Sektor der eingebetteten Systeme wie Smart-TV und Settop-Boxen genannt, wird bei der Produktion ein eindeutiger Identifier, ein dazugehöriger Service Provider sowie ein Schlüssel im Gerät abgelegt. Die Bezeichnung *Gerät* beschreibt hier die Kombination aus User Agent und spezieller Hardware. Diese Zuordnung wird vorgenommen, da nicht spezifiziert ist, wo diese Schlüssel abgelegt werden, denkbar wäre eine Speicherung im User Agent oder einem Hardwaremodul. Dieser Schlüssel kann sowohl ein symmetrischer Schlüssel als auch ein private/public-Schlüsselpaar sein. Der Identifier, der Name des Schlüssels und der symmetrische Schlüssel bzw. der öffentliche Schlüssel selbst werden dem Service Provider durch den Hersteller zur Verfügung gestellt. Für jeden unterstützen Service Provider muss ein eigener Schlüssel bereitgestellt werden. Diese Schlüssel werden auch als *named origin-specific pre-provisioned keys* bezeichnet.

[1] netflix, WebCrypto Working Group auf Anfrage

- Der eigentliche Schlüssel ist unabhängig vom Identifier und Service Provider. Die Zuordnung zu einem Service Provider ist in erster Linie dazu gedacht zu evaluieren, ob der Identifier mitgeteilt werden muss. Der Schlüssel selber ist, wie auch aus der Beschreibung den `Key-Interface` ersichtlich ist, völlig unabhängig von einem Service Provider.

Die Schlüsselsuche ist in Abb. 3.4 dargestellt.

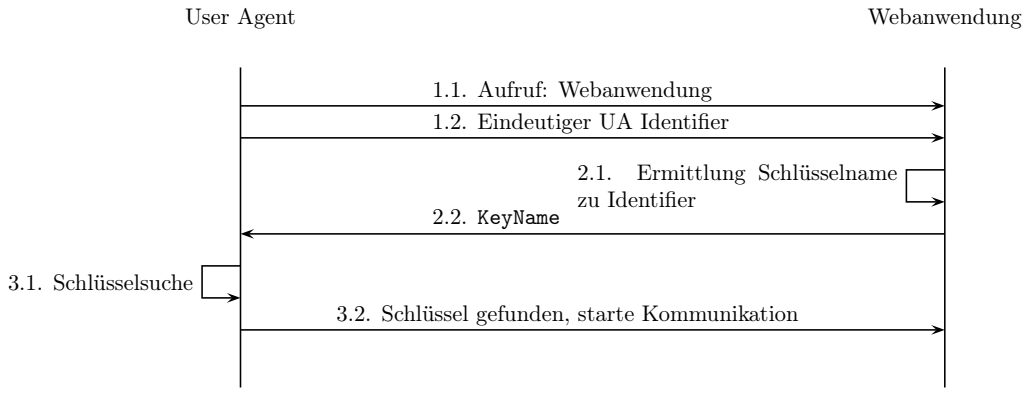


Abbildung 3.4.: Key Discovery, [3]

1. Der User Agent ruft die Webanwendung auf und teilt ihr den eindeutigen Identifier mit. Für die Mitteilung des Identifiers ist ein speziell angepasster User Agent nötig, der den Identifier anhand der angeforderten Webanwendung übermittelt. Es sollten für jede Webanwendung ein eigenes Schlüssel/Identifier-Paar angelegt werden um das Tracking zu erschweren. Wie die Zuordnung von Webanwendung und Identifier realisiert werden soll, ist nicht spezifiziert.

2. Im nächsten Schritt sucht die Webanwendung, unter Zuhilfenahme des Identifiers, den passenden Schlüssel und den dazugehörigen Namen. Dieser muss dem im User Agent hinterlegten entsprechen. Nach der Ermittlung des Namens wird dieser an den User Agent zurück geschickt.

3. Mit dem übermittelten Namen führt der User Agent nun die Schlüsselsuche durch. Diese ist in Abb. 3.3 dargestellt. Nachdem der Schlüssel gefunden ist, kann die Kommunikation beginnen.

3.4.2.3.5. Gefahren Die sogenannten *named origin-specific pre-provisioned keys* bringen allerdings auch Probleme mit sich, die in [8, Abschn. 5.1.2] benannt werden, da sie über die gesamte Lebensdauer des User Agents und damit des Gerätes verwendet werden. So ist es möglich, Benutzer mit Hilfe dieser Schlüssel identifizieren. Dies kann geschehen, wenn der Service Provider selber Tracker auf anderen Websites betreibt oder anderen Serviceprovidern anbietet, die Identifikation für sie zu übernehmen. Auf diese Weise können Nutzungsprofile über die gesamte Lebensdauer eines Geräts erstellt werden. Um dies zu verhindern werden in [8, Abschn. 5.1.2] einige Sicherheitsmechanismen vorgeschlagen:

- Der User muss explizit zustimmen, bevor ein Schlüssel verwendet werden darf. Dieser Ansatz allerdings den Nachteil, dass der Benutzer Entscheidungen bezüglich der Sicherheit treffen muss, was sich laut [6], in der Vergangenheit bereits als problematischer Punkt erwiesen hat.
- Nur Scripte, die direkt von der angeforderten Webanwendung stammen, dürfen auf die Schlüssel zugreifen. Dadurch würde der Implementierung von Trackern entgegengewirkt.
- Dem Benutzer muss klar gemacht werden, in welcher Form diese Schlüssel verwendet werden können. Problematisch ist an dieser Stelle ebenfalls der User, da er dafür die verwendeten Konzepte verstehen müsste, wovon laut [6], nicht ausgegangen werden kann.
- Ein weiterer Ansatz ist das Teilen von *Blacklists*, Listen in denen aufgeführt ist, welche Webanwendungen nicht auf die Schlüssel zugreifen dürfen.

Damit diese Konzepte funktionieren, müssten die Benutzer wissen, worüber sie eine Entscheidung treffen. Dies ist denkbar, da nach [6] ein Großteil der Anwender bereits bei einer TLS/SSL Zertifikatsfehlermeldung überfordert sind.

3.4.2.4. CryptoOperation-Interface

Hinweis: Dieses Interface wurde mit dem *Editors Draft* vom 05.08.2013 aus der Spezifikation entfernt und die zu verarbeitenden Daten wurden dem Aufruf der eigentlichen Crypto-Operation hinzugefügt, wo allerdings nicht näher darauf eingegangen wird. Da das Prozessmodell nun dem Entwickler überlassen wird und in der Spezifikation nicht weiter erwähnt wird, wird dieses Interface hier beschrieben, um ein Beispiel zu geben, wie ein solches Modell aussehen könnte.

Das `CryptoOperation-Interface`, in [2, Abschn. 12] eingeführt, beschreibt den Ablauf den einer kryptografischen Operation. Es benutzt ebenfalls asynchrone Operationen unter Verwendung von `futures`, die in Anhang A.1.1 näher beschrieben werden. Die Attribute und Funktionen sind in Tab. 3.3 sowie Tab. 3.4 dargestellt. Das Prozessmodell wird in Abb. 3.5 dargestellt.

Attribut	Beschreibung
`Key` **`Key`**	Enthält den Schlüssel, wie im `Key-Interface` spezifiziert, für die kryptografische Operation. Das Feld kann `null` sein, wenn kein Schlüssel benötigt wird, wie bei der `Digest()` Funktion
`Algorithm` **`algorithm`**	Beschreibt den zur Operation gehörenden Algorithmus

Tabelle 3.3.: Member CryptoOperation-Interface, [2, Abschn. 12.2]

Methode	Eingabe	**Beschreibung**
`process()`	`ArrayBufferView` **`data`**	Fügt Daten zur *List of Pending Data* des Prozessmodells auf und führt diese aus. Das Prozessmodell ist in Abschnitt. 3.4.2.4.1 näher beschrieben
`finish()`	`void`	Weist den Resolver[2] an, das Ergebnis zurück zu geben
`error()`	`void`	Weist den Resolver an, die *List of Pending Data* zu löschen und einen Fehler zurück zu geben

Tabelle 3.4.: Methoden CryptoOperation-Interface, [2, Abschn. 12.3]

3.4.2.4.1. Prozessmodell Das Prozessmodell der Krypto-Operation, wie in [2, Abschn. 12.1] spezifiziert und in Abb. 3.5 dargestellt, durchläuft folgende Schritte:

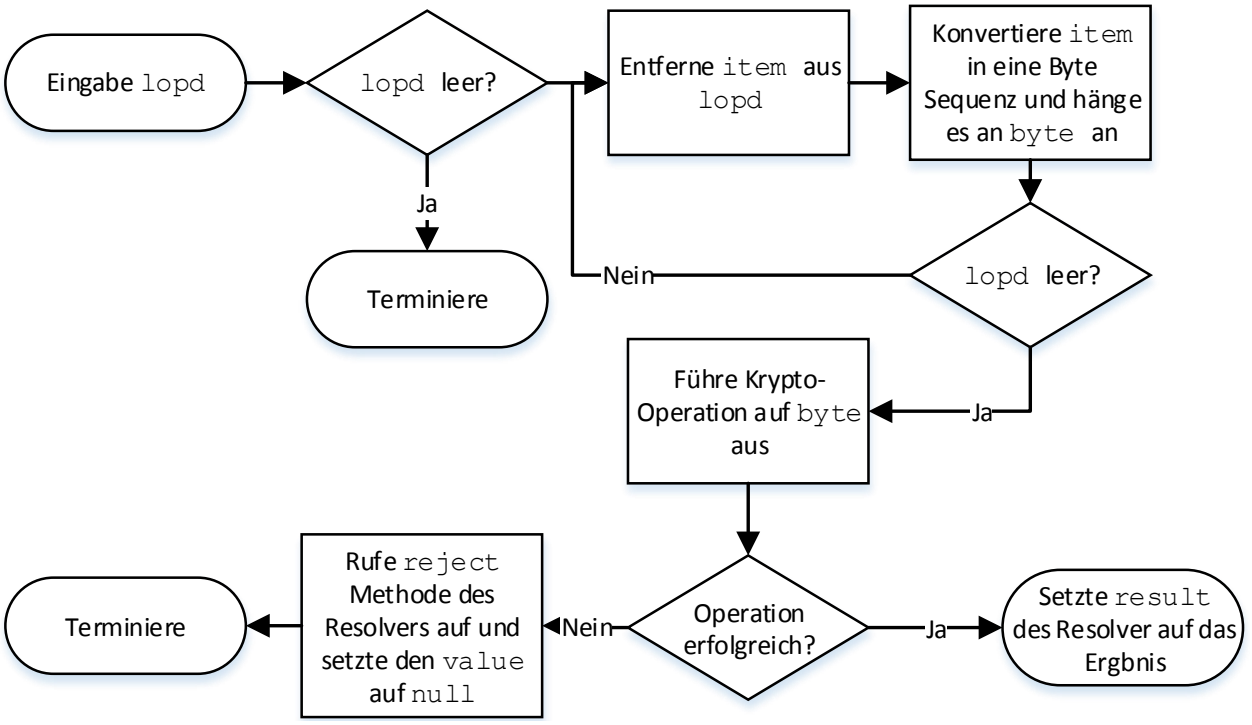


Abbildung 3.5.: CryptoOperation-Interface : Prozessmodell, cite[Abschn. 12.1]WebCrypto

Eingabe: `ListOfPendingData` im folgenden kurz `lopd` genannt. Diese wird durch das `ArrayBufferView` **`data`** der aufrufenden `process()`-Methode gefüllt. Jeder Aufruf fügt dabei ein Datensatz `item` der Liste hinzu. Die `lopd` arbeitet nach dem *First in first out* Prinzip.
Variablen: **`bytes`** ist eine Bytesequenz
Ausgabe: **`output`**, Ausgabe des zugeordneten `resolvers`.

- Zunächst wird überprüft, ob die `lopd` leer ist oder nicht. Wenn sie leer ist, wird der Algorithmus

terminiert, wenn nicht, wird die Ausführung fortgeführt.

- Das älteste `item` wird aus der `lopd` entfernt, in eine `Bytesequenz` konvertiert und an **`bytes`** angehängt.
- Wenn weitere Elemente in der `lopd` vorhanden sind, wird der vorherige Schritt wiederholt, bis die `lopd` leer ist, wenn nicht, wird die Ausführung fortgeführt.
- Die gewünschte Crypto-Operation wird auf **`bytes`** ausgeführt.
- Wenn diese Operation erfolgreich war, wird **`result`** des `resolvers` auf das Ergebnis gesetzt und die `resolver.resolve()`-Methode des `resolvers` aufgerufen. Dadurch wird das Ergebnis an die aufrufende Funktion zurück gegeben. Wenn die Operation nicht erfolgreich war, wird **`result`** des `resolvers` auf `null` gesetzt und die `resolver.reject()`-Methode des `resolvers` aufgerufen. Der Algorithmus wird anschließend terminiert.

3.4.2.5. Crypto-Interface

Das `Crypto-Interface` wird in [2, Abschn. 13] beschrieben und ist im Prinzip lediglich ein Container, welches das `Window-Interface` erweitert und Zugriff auf die Crypto-Operationen ermöglicht. Als einziges Attribut besitzt es `SubtleCrypto` **`subtle`**, eine Implementierung des `SubtleCrypto-Interface`. Im Weiteren implementiert `crypto` das `RandomSource-Interface` . Dabei ist `crypto` eine Implementierung des`Crypto-Interface`. .

3.4.2.6. SubtleCrypto-Interface

Hinweis: Durch den Wegfall des `CryptoOperation-Interface` im *Editors Draft* vom 05.08.2013 haben sich im `SubtleCrypto-Interface` Änderungen ergeben. Da dieses Interface ein Zentrales ist, wird hier die Spezifikation von 05.08.2013 zu Grunde gelegt. Die restlichen Teile der Arbeit bleiben davon unberührt.

Das `SubtleCrypto-Interface` wird in [2, Abschn. 14][3] eingeführt und ist das zentrale und auch umfangreichste Interface der API, da es die eigentlichen Crypto-Methoden zur Verfügung stellt. Alle Operationen sind in Tab. 3.6 aufgelistet. Die Methoden `wrapKey()` und `unwrapKey()` stehen noch in der Diskussion, es ist daher möglich, das diese noch entfernt werden [2, Issue-35]. Da alle Methoden ein `Future<any>`-Objekt, eine nicht typisierte Rückgabe einer asynchonenen Opertaion, als Rückgabewert verwenden, wird auf die Angabe eines Rückgabewertes in der Tabelle, aus Gründen der Übersichtlichkeit verzichtet. Die Beschreibung der `Enumeration` **`KeyFormat`** ist der Tab. 3.5 zu entnehmen.

[3] Abschn.13 im *Editors Draft* vom 05.08.2013

Enumeration	**Werte**	**Beschreibung**
`enumeration` **`KeyFormat`**	raw	Unformatierte Bytesequenz für geheime, symmetrische Schlüssel
	pkcs8	DER Encoding der Struktur für private Schlüssel. Spezifiziert in RFC 5208 [11]
	spki	DER Encoding der Struktur für SubjectPublicKeyInfo Struktur. Spezifiziert in RFC 5280 [12]
	jwk	Schlüsselrepräsentation im JSON Web Key Format

Tabelle 3.5.: Enumerations SubtleCrypto-Interface, [2, Abschn. 14] [3]

Methode	**Eingabe**	**Beschreibung**
`encrypt()`	`AlgorithmIdentifier` **`algorithm`** `Key` **`key`** `sequence<CryptoOperationData>` **`data`**	Führt eine Verschlüsselungsoperation mit dem gegebenen Schlüssel und Algorithmus auf `data` aus
`decrypt()`	`AlgorithmIdentifier` **`algorithm`** `Key` **`key`** `sequence<CryptoOperationData>` **`data`**	Führt eine Entschlüsselungsoperation mit dem gegebenen Schlüssel und Algorithmus auf `data` aus
`sign()`	`AlgorithmIdentifier` **`algorithm`** `Key` **`key`** `sequence<CryptoOperationData>` **`data`**	Führt eine Signaturoperation mit dem gegebenen Schlüssel und Algorithmus auf `data` aus
`verify()`	`AlgorithmIdentifier` **`algorithm`** `Key` **`key`** `CryptoOperationData` **`signature`** `sequence<CryptoOperationData>` **`data`**	Führt eine Signaturvalidierung mit dem gegebenen Schlüssel und Algorithmus auf `data` aus
`digest()`	`AlgorithmIdentifier` **`algorithm`** `sequence<CryptoOperationData>` **`data`**	Führt eine Hashoperation mit dem gegebenen Algorithmus auf `data` aus

`generateKey()`	`AlgorithmIdentifier` **`algorithm`**
	`optional bool` **`extractable`** `= false`
	`optional KeyUsage` **`keyUsage`** `= []`
		Erzeugt einen Schlüssel für den angegebenen Algorithmus und legt ihm im Schlüsselspeicher ab

deriveKey()	AlgorithmIdentifier **derivedKeyType** Key **baseKey** optional bool **extractable** = false optional KeyUsage **keyUsage** = []	Leitet einen Schlüssel aus dem gegebenen Schlüssel für den gegebenen Algorithmus ab
deriveBits()	AlgorithmIdentifier **algorithm** Key **baseKey** unsigned long **lenght**	Leitet eine angegebene Menge von Bits aus dem gegebenen Schlüssel für den gegebenen Algorithmus ab
importKey()	KeyFormat **format** CryptoOperationData **KeyData** AlgorithmIdentifier **algorithm** optional bool **extractable** = false optional KeyUsage **keyUsage** = []	Importiert den gegebenen Schlüssel mit den angegebenen Werten
exportKey()	KeyFormat **format** Key **key**	Exportiert den angegebenen Schlüssel in das spezifizierte Format
wrapKey()	KeyFormat **format** Key **key** Key **wrappingKey** AlgorithmIdentifier **algorithm**	Exportiert den angegebenen Schlüssel **key** und verschlüsselt ihn mit dem angegebenen Schlüssel **wrappingKey** und dem spezifizierten Algorithmus
unwrapKey()	KeyFormat **format** CryptoOperationData **wrappedKey** Key **unwrappingKey** AlgorithmIdentifier **uwrapAlgorithm** AlgorithmIdentifier **unwrapedKeyAlgorithm** optional bool **extractable** = false optional KeyUsage **keyUsage** = []	Entschlüsselt den in **wrappedKey** übergebenen Schlüssel mit dem angegebenen Algorithmus und dem Schlüssel **unwrappingKey** und weist ihm die übergebenen Parameter zu

Tabelle 3.6.: Methoden SubtleCrypto-Interface, [2, Abschn. 14] [3]

Der Ablauf aller Methoden erfolgt nach dem im [2, Abschn. 14.2] [3] spezifizierten und in Abb. 3.6 dargestellten Schema, einzig die Eingabewerte unterscheiden sich. Die funktionsspezifischen Eingabewerte können Tab. 3.6 entnommen werden, die Aufschlüsselung der verschiedenen Eingabewerte ist in Tab. 3.7 darge-

stellt. Die unterschiedliche Funktionalität wird durch die verschiedenen Methoden des User Agent erreicht. Die Methoden `importKey()` und `exportKey()` sind nicht spezifiziert. Diese sollten allerdings auch nicht direkt durch JavaScript aufgerufen werden, da dabei Schlüsselmaterial an JavaScript übergeben werden müsste. Eine Ausnahme bilden hierbei Schlüssel, die als `public` gekennzeichnet sind. Schlüssel die als `public` gekennzeichnet werden sind Schlüssel, die den öffentlichen Teil eines Schlüsselpaares darstellen. Wenn ein solches Schlüsselpaar erzeugt wird, kann der öffentliche Teil an die Webanwendung weitergegeben werden, damit diese Signaturen, die vom User Agent erzeugt wurden validieren kann oder Daten asymmetrisch verschlüsselt an den User Agent übertragen kann.

Parameter	**Beschreibung**
`AlgorithmIdentifier` **`algorithm`**	Gibt den Algorithmus an, der für die Krypto-Operation verwendet werden soll
`Key` **`key`**	Gibt den Schlüssel an, der für die Krypto-Operation verwendet werden soll
`sequence<CryptoOperationData>` **`data`**	Dies sind die Daten, auf denen die Krypto-Operation durchgeführt werden soll
`CryptoOperationData` **`signature`**	Im Falle einer Singnaturüberprüfung wird in diesem Parameter die Signatur, gegen die geprüft werden soll, übergeben
`optional bool` **`extractable`** `= false`	Dieser Wert bezieht sich auf die Parameter, des Key-Interface, welche in Tab. 3.2 beschrieben sind. Standardwert sollte *false* sein
`optional KeyUsage` **`keyUsage`** `= []`	Dieser Wert bezieht sich auf die Parameter, des `Key-Interface`, welche in Tab. 3.2 beschrieben sind. Er kann die Werte annehmen, die in Tab. 3.1 spezifiziert sind
`AlgorithmIdentifier` **`derivedKeyType`**	Gibt den Algorithmus an, für den Schlüssel abgeleitet werden sollen
`Key` **`baseKey`**	Gibt an, von welchem Schlüssel weitere Schlüssel bzw. Bits abgeleitet werden sollen
`unsigned long` **`lenght`**	Gibt die Anzahl an Bytes an, die abgeleitet werden sollen
`KeyFormat` **`format`**	Bestimmt das Format eines Schlüssels. Mögliche Formate können Tab. 3.5 entnommen werden
`Key` **`wrappingKey`**	Gibt an, mit welchem Schlüssel weitere Schlüssel verschlüsselt werden sollen
`CryptoOperationData` **`wrappedKey`**	Repräsentiert einen verschlüsselten Schlüssel, der entschlüsselt werden soll

`Key` **`unwrappingKey`**	Gibt an, mit welchem Schlüssel ein verschlüsselter Schlüssel entschlüsselt werden soll
`AlgorithmIdentifier` **`unwrapAlgorithm`**	Gibt an, mit welchem Algorithmus ein verschlüsselter Schlüssel entschlüsselt werden soll
`AlgorithmIdentifier` **`unwrapedKeyAlgorithm`**	Gibt an, zu welchem Algorithmus ein verschlüsselter Schlüssel nach der Entschlüsselung gehört

Tabelle 3.7.: Eingabeparameter SubtleCrypto-Interface, in Reihenfolge des Auftretens in Tab. 3.6

Um eine Operation durchzuführen, müssen folgende Schritte durchlaufen werden:

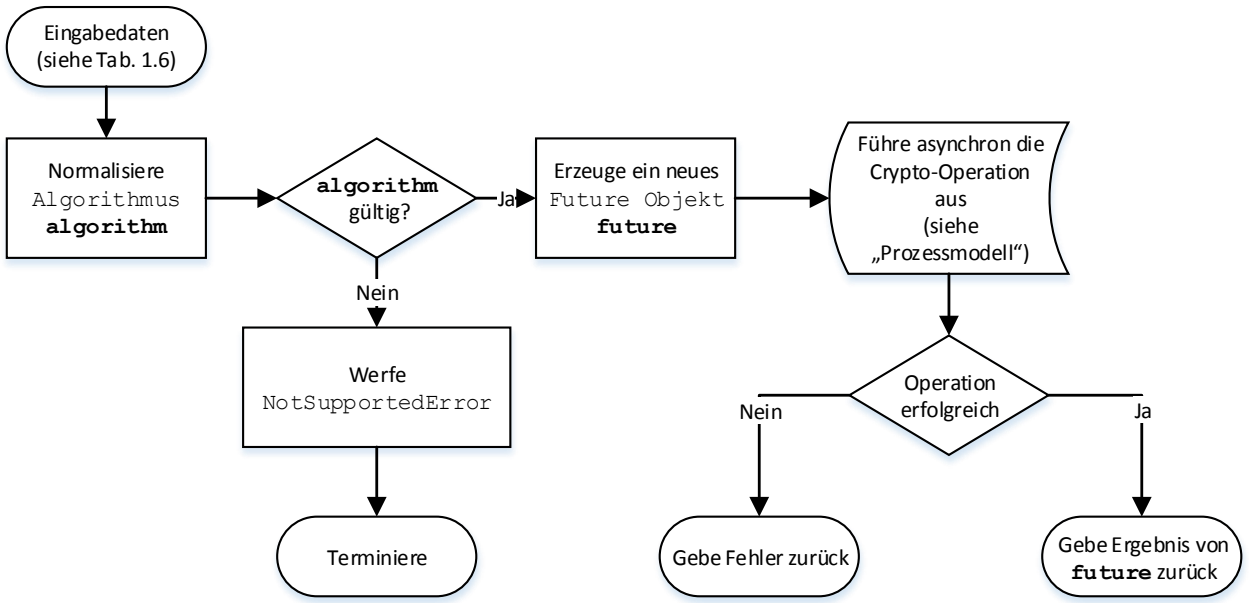


Abbildung 3.6.: Ablaufdiagramm der Methoden, [2, Abschn. 14.2]

Eingabe: Ist Tab. 3.7 zu entnehmen
Ausgabe: `Future-Objekt`

- **`algorithm`** wird nach den in Abschnitt 3.4.3.1 beschriebenen *algorithm normalizing rules* zu **`normalizedAlgorithm`** normalisiert.
- Wenn **`normalizedAlgorithm`** registriert ist, wird die Ausführung fortgesetzt, sonst wird ein `NotSupportedError` geworfen und der Vorgang terminiert.
- Ein neues `Future-Objekt` **`future`** wird erzeugt und ausgeführt. Die Crypto-Operation wird

nun vom `resolver` des **`future`** durchgeführt. Ein Beispiel für eine Verarbeitung ist in Abschnitt 3.4.2.4.1 und Abb. 3.5 gegeben.

- Wenn die Operation erfolgreich ist, wird das Ergebnis der Crypto-Operation von **`future`** zurückgegeben. Sonst wird eine Fehlermeldung zurückgegeben. Die dafür benötigten Methoden `resolver.resolve()` bzw. `resolver.reject()` werden innerhalb des Prozessmodels aufgerufen. An dieser Stelle werden lediglich die Ergebnisse abgefragt.

Trotz des gleichen Ablaufes sind bei verschiedenen Methoden verschiedene Voraussetzungen zu erfüllen. In Tab. 3.8 sind die Besonderheiten der einzelnen Methoden aufgelistet, die restlichen Anforderungen bleiben davon allerdings unberührt.

Methode	**Voraussetzung**
`deriveKey()`	Das `Key`-Objekt muss explizit den Parameter `derive` angegeben haben [2, Abschn. 14.2.7] [3]
`wrapKey()`	Das `Key`-Objekt, mit dem verschlüsselt werden soll, muss explizit den Parameter `wrap` angegeben haben. Der zu extrahierende Schlüssel muss das Attribut `extractable=true` besitzen [2, Abschn. 14.2.10] [3]
`unwrapKey()`	Das `Key`-Objekt, mit dem entschlüsselt werden soll, muss explizit den Parameter `wrap` angegeben haben. Der zu extrahierende Schlüssel muss das Attribut `extractable=true` besitzen [2, Abschn. 14.2.11] [3]

Tabelle 3.8.: Methoden CryptoInterface: Besonderheiten, [2, Abschn. 14.2]

3.4.2.7. BigInteger

Der in [2, Abschn. 16] vorgestellte `BigInterger`, stellt einen Datentyp zur Verfügung, um große Zahlen darstellen zu können. Dazu verwendet er ein Array aus 8-Bit Integerwerten.

3.4.2.8. KeyPair-Interface

Das `KeyPair-Interface` wird in [2, Abschn. 17] eingeführt und stellt ein Interface zu Verfügung, um Paare von öffentlichen und privaten Schlüsseln darzustellen. Die Member des Interface sind in Tab. 3.9 gegeben.

Attribut	Beschreibung
`Key` **`publicKey`**	Repräsentiert den öffentlichen Schlüssel
`Key` **`privateKey`**	Repräsentiert den privaten Schlüssel

Tabelle 3.9.: Member KeyPairInterface, [2, Abschn. 17]

3.4.2.9. WorkerCrypto-Interface

Ermöglicht Scripten, die im Hintergrund laufen, den Zugriff auf Crypto-Operationen. Diese Hintergrundprozesse werden in im *WebWorkers Standard* [10] spezifiziert.

3.4.3. Algorithmen

Da die Implementierung von Crypto-Algorithmen nicht Bestandteil des Standards ist, gibt es auch keine vorgeschriebenen Algorithmen, die ein User Agent unterstützen muss. Eine Übersicht, welche Algorithmen für welche Operationen verwendet werden können, gibt Tab. 3.10. Die hervorgehobenen Algorithmen werden explizit empfohlen, um eine gewisse Interoperabilität zu erreichen. Da laut [2, ISSUE-35] noch nicht feststeht, ob `wrapKey()` und `unwrapKey()` in die endgültige Spezifikation aufgenommen werden, existieren auch noch keine Algorithmenempfehlungen für diese Funktionen.

Algorithmus	`enc`	`dec`	`sig`	`ver`	`dig`	`gen`	`dK`	`dBi`	`imp`	`exp`
RSAES-PKCS1-v1_5	✓	✓				✓			✓	✓
RSASSA-PKCS1-v1_5			✓	✓		✓			✓	✓
RSA-PSS			✓	✓		✓			✓	✓
RSA-OAEP	✓	✓				✓			✓	✓
ECDSA			✓	✓		✓			✓	✓
ECDH						✓	✓	✓	✓	✓
AES-CTR	✓	✓				✓			✓	✓
AES-CBC	✓	✓				✓			✓	✓
AES-CMAC			✓	✓		✓			✓	✓
AES-GCM	✓	✓				✓			✓	✓
AES-CFB	✓	✓				✓			✓	✓
HMAC			✓	✓		✓			✓	✓
DH						✓	✓	✓	✓	✓
SHA-1 / 224 / 384 / 512					✓					
SHA-256					✓					
CONCAT							✓	✓		
HKDF-CTR							✓	✓		
PBKDF2							✓	✓		

Abkürzungen:

enc : `encrypt()` **ver** : `verify()` **dK** : `deriveKey()` **exp** : `exportKey()`
dec : `decrypt()` **dig** : `digest()` **dB** : `deriveBit()`
sig : `sign()` **gen** : `generateKey()` **im** : `importKey()`

Tabelle 3.10.: Algorithmen, [2, Abschn. 17.1]

Grundsätzlich ist die API darauf ausgelegt, dass jeder Entwickler eigene Crypto-Algorithmen entwickeln und registrieren kann. Um einen Algorithmus in der API zu registrieren müssen im, in [2, Abschn. 10] vorgestellten `Algorithm Dictionary`, folgende Werte hinterlegt werden:

- Ein *Canonical Name* für den Algorithmus muss bestimmt werden. Dieser darf nur ASCII Zeichen enthalten und muss im `LowerCase` eindeutig von jedem anderen *Canonical Name* und *Alias* unterscheidbar sein.
- Es müssen sämtliche Operationen definiert und implementiert werden.
- Alle Übergabeparameter müssen definiert werden.
- Die Ausgabe muss für jeden Algorithmus definiert werden.
- Jeder Algorithmus sollte einen oder mehrere *Alias* definieren. Für den *Alias* gelten die selben Regeln wie für den *Canonical Name*.

3.4.3.1. Normalisierung

Damit ein Algorithmus eindeutig bestimmt werden kann, muss dieser zunächst einmal, wie in [2, Abschn. 19] spezifiziert und in Abb. 3.7 gezeigt, normalisiert werden. Dazu müssen folgende Schritte durchgeführt werden:

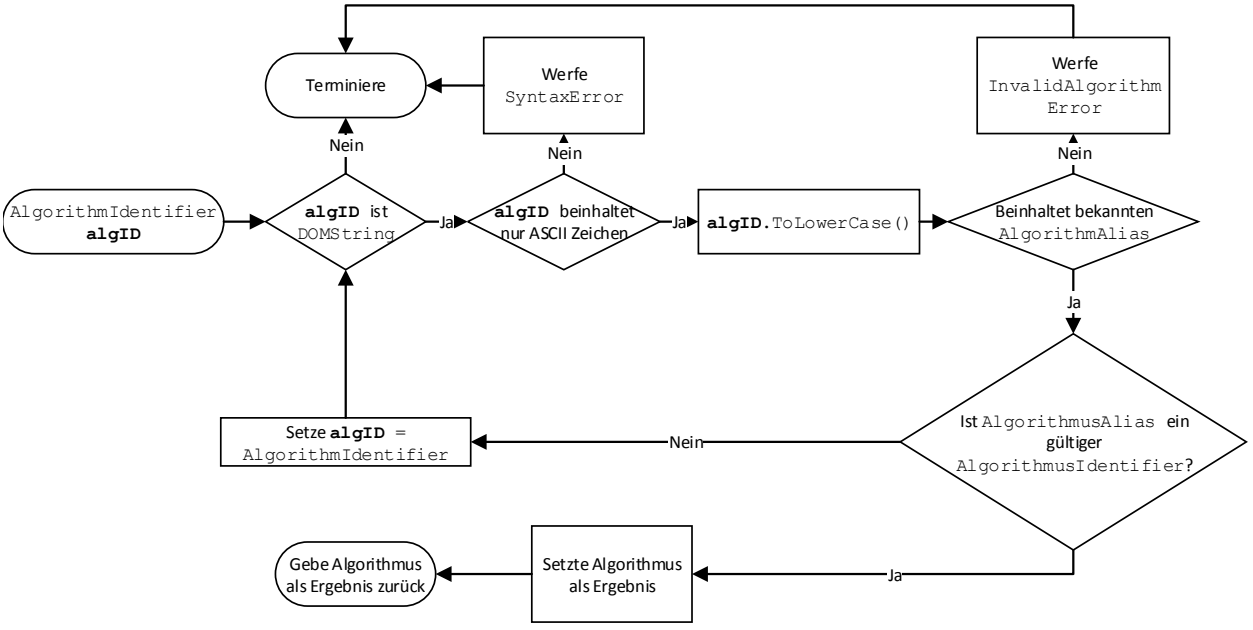


Abbildung 3.7.: Algorithmus Normalisierung, [2, Abschn. 19]

Eingabe: `AlgorithmIdentifer` **`algID`**
Ausgabe: `AlgorithmDictiornay` **`dict`**

- **`algID`** wird übergeben und es wird geprüft, ob es sich um einen `DOMString` handelt, wenn dies nicht der Fall ist, wird der Vorgang terminiert, sonst fortgesetzt.
- Wenn **`algID`** nur ASCII Zeichen beinhaltet, wird dieser in den *LowerCase* transformiert, dabei werden alle Buchstaben in Kleinbuchstaben umgewandelt. Wenn nicht, wird ein `SyntaxError` geworfen und der Vorgang terminiert.
- Es wird überprüft, ob **`algID`** einen bekannten `AlgorithmAlias` beinhaltet, wenn dies nicht der Fall ist, wird ein`InvalidAlgorithmError` geworfen und der Vorgang terminiert. Sonst wird geprüft, ob der `AlgorithmAlias` einem gültigen `AlgorithmIdentifier` entspricht. Wenn dies der Fall ist, dann wird das `AlgorithmDictionary` zu dem Identifier ermittelt und als Ergebnis übergeben. Wenn nicht, wird der Algorithmus mit dem `AlgorithmAlias` als **`algID`** neu gestartet.

4. Spezifizierte Anwendungsfälle

Im Rahmen der Spezifikation sind einige Use Cases aufgeführt, die nun beschrieben und anhand von Beispielen verdeutlicht werden sollen. Alle Anwendungsfälle werden so dargestellt, wie sie in der Spezifikation [2] und den UseCases [13] beschrieben werden. Die Abläufe wurden nicht gezielt auf mögliche Angriffe überprüft.
In allen folgenden Use Cases wird davon ausgegangen, dass:

- die Webanwendung in der Lage ist, den Namen des zu verwendenden Schlüssels zu ermitteln. Die Ermittlung des Schlüsselnamens ist in Abschnitt 3.4.2.3.4 beschrieben. Ebenso wurden evtl. benötigte öffentliche Schlüssel vorher sicher ausgetauscht.
- die Verbindung durch TLS/SSL gesichert ist und alle Zertifikate gültig sind.
- übermittelte, öffentliche Schlüssel, aufgrund der TLS/SSL-Verbindung authentisch sind.
- die Webanwendung und der User Agent die passenden Algorithmen implementieren.
- wenn ein Schlüssel über die API erzeugt wird, wird diese im Schlüsselspeicher des User Agent hinterlegt, auch wenn dies nicht explizit angegeben ist.
- Anmeldeverfahren in den Diagrammen aus Gründen der Übersichtlichkeit nicht dargestellt sind, sondern werden nur durch den Hinweis „Anmeldung“ deutlich gemacht.
- wenn Signaturen übertragen werden, werden die dazugehörigen Daten im Klartext ebenfalls übertragen, auf die explizite Aufzählung wurde jedoch aus Gründen der Übersichtlichkeit verzichtet.
- alle beschriebenen Anwendungsfälle fehlerfrei durchgeführt werden.

Es wird in jedem Anwendungsfall auf die verwendeten Funktionalitäten verwiesen. Dazu wird die in Tab. 4.1 erläuterte Schreibweise verwendet. Zu Beginn jedes Anwendungsfalles werden die Funktionalitäten aufgelistet, die im kompletten Anwendungsfall verwendet werden. Ergänzend dazu werden in jedem Schritt die verwendeten Methoden genannt. Bei manchen Funktionalitäten wird explizit zwischen symmetrischen und asymmetrischen Anwendungen unterschieden. Diese Unterscheidung dient lediglich der Übersichtlichkeit, in den eigentlichen Methoden wird die Entscheidung, ob eine Operation mit symmetrischen oder asymmetrischen Mitteln durchgeführt wird, über den verwendeten Algorithmus angegeben. Die Funktionen werden sowohl für Client- als auch Serverseitige Operationen angegeben.

Bezeichnung	**Beschreibung**
[DIGEST]	Erzeugung eines kryptografischen Hash-Wertes
[MAC]	Erzeugung eines *Message Authentication Codes*
[HMAC]	Erzeugung eines *Hash-Wertes*, in den bei der Erzeugung ein Schlüssel eingebunden wird
[SIGN]	Erzeugung einer digitalen Signatur
[VERIFY]	Verifizierung eines MAC, HMAC oder einer digitalen Signatur
[ENCRYPT]	Durchführung einer Verschlüsselung
[DECRYPT]	Durchführung einer Entschlüsselung
[DERIVE-SYM]	Durchführung einer Schlüsselableitung für symmetrischen Algorithmen
[DERIVE-ASSYM]	Durchführung einer Schlüsselableitung für asymmetrischen Algorithmen
[KEYGEN-SYM]	Erzeugung eines Schlüssel für symmetrischen Algorithmen
[KEYGEN-ASSYM]	Erzeugung eines Schlüssel für asymmetrischen Algorithmen
[IMPORT]	Import eines Schlüssels oder eines Schlüsselpaares
[EXPORT]	Export eines Schlüssels oder eines Schlüsselpaares
[KEYEX]	Aushandlung bzw. Austausch von Schlüsseln
[KEYEX-DH]	Aushandlung bzw. Austausch von Schlüsseln. Hier wird ausdrücklich der Diffie-Hellmann key exchange gefordert
[KEYCALL]	Möglichkeit der Webanwendung, einen symmetrischen Schlüssel in der eigenen Schlüsselverwaltung zu finden
[RANDOM]	Erzeugen einer Zufallszahl
[JWK]	Möglichkeit einen öffentlichen Schlüssel im *JSON Web Key format* (JWK) darzustellen
[WRAP]	Verschlüsselung eines Schlüssels mit einem anderen Schlüssel
[UNWRAP]	Entschlüsselung eines Schlüssels mit einem anderen Schlüssel
[NAMEDKEY]	Ermöglicht die Suche nach bereits vorhandenen Schlüsseln im Schlüsselspeicher des User Agent

Tabelle 4.1.: Funktionalitäten, [13, Abschn. 1]

4.1. Multi-Factor Authentication

Ziel: Ein Benutzer authentifiziert sich zusätzlich zu dem Benutzernamen und Kennwort mit einem Schlüssel.

Verwendete Funktionalitäten: ***[RANDOM]***, ***[VERIFY]***, ***[DIGEST]***, ***[SIGN]***, ***[NAMEDKEY]***, ***[KEYCALL]***

Voraussetzungen: Beide Parteien verfügen über ein Schlüsselpaar. Der öffentliche Schlüssel ist der anderen Partei bekannt.

Ablauf: Der Benutzer meldet sich mit mit seinem Benutzernamen und Kennwort an eine Webanwendung an. Um nun aber sicher zu stellen, dass es sich in der Tat um den richtigen Benutzer handelt, muss dieser zusätzlich beweisen, dass er im Besitz eines Schlüssels ist. So müsste ein Angreifer nicht nur den Benutzernamen und das Passwort eines Anwenders ausspähen, sondern müsste auch noch Kenntnis über den privaten Schlüssel des Anwenders haben, um sich an die Webanwendung anmelden zu können. Dieser Anwendungsfall wird in [2, Abschn. 2.1] nur als möglicher Anwendungsfall genannt, ohne das näher darauf eingegangen wird. In diesem Beispiel wird dafür ein einfaches Authentifikations-Protokoll verwendet. Diese Protokollklasse ist sowohl für symmetrische als auch asymmetrische verfügbar, daher liegt es in der Hand des Entwicklers für welche Art Schlüssel er sich entscheidet. In Abb. 4.1 ist als Beispiel das *ISO/IEC 9798-3 threepass mutual authentication protocol* gegeben, das asymmetrisches Kryptographie verwendet und in [7] näher beschrieben ist.

Nachdem sich der Benutzer mit Hilfe seines Benutzernamens und Passwortes angemeldet hat, startet die Webanwendung die Prozedur, in der der Benutzer beweist, dass er im Besitz eines bestimmten Schlüssels ist. Dazu führt die Webanwendung folgende Schritte aus:

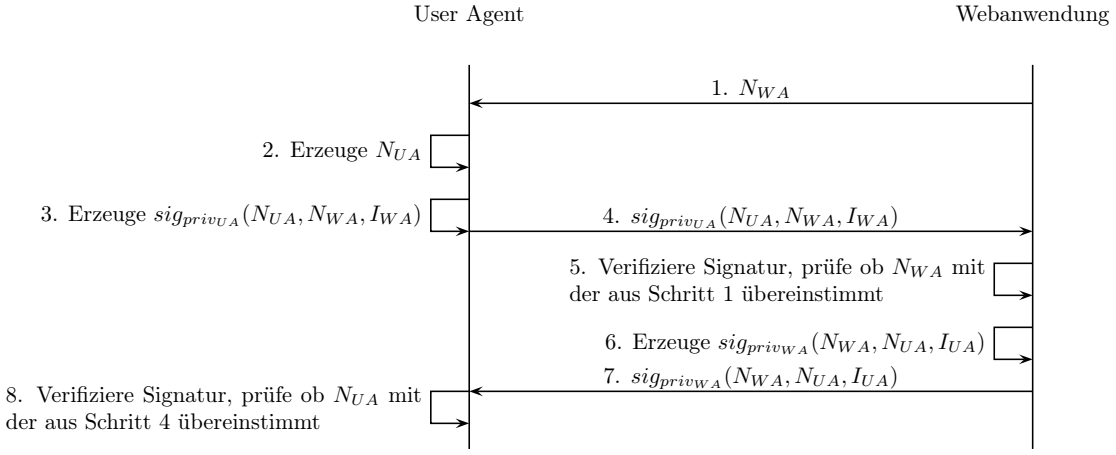


Abbildung 4.1.: ISO/IEC 9798-3 threepass mutual authetication protocol [7]

1. Erzeugen einer Nonce, einer Zufallszahl, N_{WA}. Diese wird an den User Agent übertragen. ***[RANDOM]***

2. Erzeugen einer Nonce N_{UA}. Dies geschieht mit der `getRandomValue()` Methode der API. ***[RANDOM]***

3. Erzeugen einer Signatur $sig_{priv_{UA}}(N_{UA}, N_{WA}, I_{WA})$ über die Werte N_{WA}, N_{UA} und I_{WA} mit dem privaten Schlüssel $priv_{UA}$ des User Agent. I_{WA} ist ein Identifier, der die Webanwendung identifiziert. Dieser Identifier kann z.B. die IP Adresse oder der Hostname der Webanwendung sein. Dieser ist aber protokollspezifisch und dient alleine der Sicherheit des Protokolls. Dieser wird, in diesem Szenario von der Webanwendung, zusammen mit dem JavaScript für den Authentifizierungsvorgang übertragen. ***[SIGN]***, ***[DIGEST]***, ***[NAMEDKEY]***

4. Die Signatur $sig_{priv_{UA}}(N_{UA}, N_{WA}, I_{WA})$ wird zusammen mit den Werten N_{WA}, N_{UA} und I_{WA} im Klartext an die Webanwendung übertragen.

5. Die Webanwendung verifiziert nun die Signatur und prüft, ob die eben empfangene Nonce N_{WA} mit der aus Schritt 1 übereinstimmt. Wenn eine dieser Prüfungen fehlschlägt, wird der Vorgang abgebrochen, da entweder Teile der Nachricht verändert wurden und so die Signatur ungültig wurde oder eine alte Nachricht erneut gesendet wurde, was unterschiedliche Werte für N_{WA} zu folge hätte. ***[VERIFY]***, ***[KEYCALL]***

6. Die Webanwendung erzeugt nun mit ihrem privaten Schlüssel $priv_{WA}$ die Signatur $sig_{priv_{WA}}(N_{WA}, N_{UA}, I_{UA})$ über die Werte N_{WA}, N_{UA} und I_{UA}. ***[DIGEST]***, ***[SIGN]***

7. Die Signatur wird zusammen mit den Werten N_{UA}, N_{WA} und I_{UA} im Klartext als $sig_{priv_{WA}}(N_{WA}, N_{UA}, I_{UA}), N_{WA}, N_{UA}, I_{UA}$ an den User Agent übertragen.

8. Der User Agent verifiziert nun die Signatur und prüft, ob die eben empfangene Nonce N_{UA} mit der aus Schritt 2 übereinstimmt. ***[VERIFY]***, ***[NAMEDKEY]***

4.2. Protected Document Exchange

Ziel: Wie in [2, Abschn. 2.2] spezifiziert und in Abb. 4.2 dargestellt, möchte eine Arbeitsgruppe auf vertrauliche Dokumente gemeinsam zugreifen. Diese Dokumente sollen verschlüsselt in einer Webanwendung hinterlegt werden. Bisher muss man die Dokumente zunächst in einer externen Anwendung verschlüsseln und dann das verschlüsselte Dokument hochladen. Der Austausch des Schlüssels zum entschlüsseln des Dokumentes muss ebenfalls vom Anwender übernommen werden. Die API soll dies dem Anwender nun abnehmen.
Verwendete Funktionalitäten: ***[KEYGEN-SYM]***, ***[WRAP]***, ***[ENCRYPT]***, ***[NAMEDKEY]***, ***[UNWRAP]***, ***[DECRYPT]***
Voraussetzungen: Jedes Mitglied der Arbeitsgruppe hat ein Schlüsselpaar und alle Mitglieder kennen alle öffentlichen Schlüssel der anderen Teilnehmer.

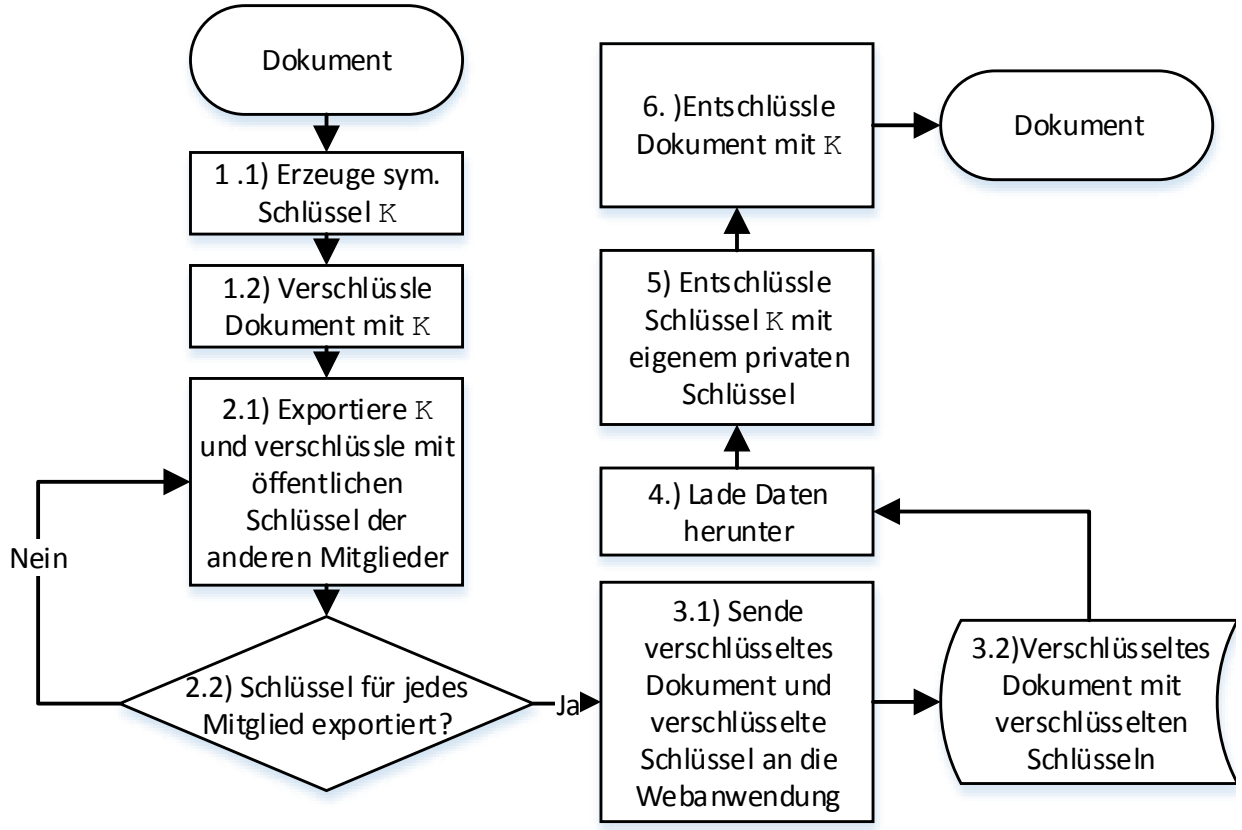


Abbildung 4.2.: Protected Document Exchange, [2, Abschn. 2.2]

Vorgehen:

1. Erzeuge einen neuen Schlüssel K und verschlüssele das Dokument mit K. ***[KEYGEN-SYM]***, ***[ENCRYPT]***
2. Exportiere K mit Hilfe der `wrap()` Methode und dem öffentlichen Schlüssel des Arbeitsgruppenmitglieds. Wiederhole den Vorgang für jedes Mitglied. ***[WRAP]***
3. Das verschlüsselte Dokument wird, zusammen mit allen verschlüsselten Schlüsseln, in der Webanwendung hinterlegt.

Zu diesem Zeitpunkt ist das verschlüsselte Dokument in der Webanwendung verfügbar und kann von jedem autorisierten Mitglied genutzt werden. Um das Dokument nutzen zu können, müssen folgende Schritte durchgeführt werden:

3. Herunterladen des Dokumentes zusammen mit allen verschlüsselten Schlüsseln.
4. Der verschlüsselte Schlüssel wird mit dem eigenen privaten Schlüssel entschlüsselt. Dazu wird die `unwrap()` Methode verwendet. ***[NAMEDKEY]***, ***[UNWRAP]***,

5. Mit dem nun vorliegenden Schlüssel K wird das Dokument entschlüsselt. Das Dokument liegt nun wieder im Klartext vor. ***[DECRYPT]***

Wie die Zuordnung der einzelnen verschlüsselten Schlüssel zu den Nutzern erfolgt ist nicht weiter beschrieben und bleibt dem Entwickler überlassen. Ein Dictionary mit einem Fingerprint der einzelnen öffentlichen Schlüssel könnte für diese Zuordnung verwendet werden. Der gesamte Vorgang wurde in Abb. 4.2 visualisiert.

4.2.1. Cloud Storage

Der Anwendungsfall *Cloud Storage* ist eine Vereinfachung des *Protected Document Exchange* und wird in [2, Abschn. 2.3] eingeführt. Dafür wird angenommen, dass nur die Person, die das Dokument erstellt und in der Webanwendung speichert, berechtigt ist, dieses auch wieder zu entschlüsseln. Aufgrund dieser Voraussetzung muss der Schlüssel K nicht exportiert werden und auch nicht in verschlüsselter Form in der Webanwendung vorgehalten werden. Im Vergleich zum *Protected Document Exchange* entfallen die Schritte 2.1, 2.2 und 5. aus .

4.3. Document Signing

Ziel: Wie in [2, Abschn.2.4] vorgeschlagen, soll ein Dokument aus einer Webanwendung digital signiert werden. Dies soll die eigenhändige Unterschrift ersetzen. Auch hier muss das Dokument bisher in einer externen Anwendung signiert werden. Durch die API soll dieser Vorgang auch komplett im User Agent stattfinden.
Verwendete Funktionalitäten: ***[SIGN]***, ***[DIGEST]***, ***[NAMEDKEY]***, ***[VERIFY]***, ***[KEYCALL]***
Voraussetzungen: Der Benutzer hat einen asymmetrischen Schlüssel, der eine eindeutige Zuordnung zulässt.
Vorgehen:

- Die Webanwendung sendet ein Dokument, z.B. eine Vertrag, an den User Agent. Dieses Dokument kann von der Webanwendung vor der Übertragung signiert werden um Manipulationen vorzubeugen. ***[SIGN]***, ***[DIGEST]***, ***[KEYCALL]***
- Wenn das Dokument signiert ist, prüfe die Signatur und signiere das Dokument mit dem geforderten Schlüssel. Wenn das Dokument nicht von der Webanwendung signiert wurde, signiere das Dokument direkt. ***[NAMEDKEY]***, ***[VERIFY]***, ***[DIGEST]***, ***[SIGN]***
- Das signierte Dokument wird an die Webanwendung zurückgeschickt. Dort wird die Signatur geprüft. ***[KEYCALL]***, ***[DIGEST]***, ***[VERIFY]***

4.4. Secure Messaging

Ziel: Zwei User Agents sollen direkt miteinander sicher kommunizieren können. Dazu rufen beide eine Webanwendung auf, die die nötigen JavaScript Methoden zu den User Agent überträgt. Als Protokoll wurde dafür, in [2, Abschn.2.6] ,das *Off-the-Record Messaging Protokoll* (OTR) vorgeschlagen, welches in [14] spezifiziert ist. Ziel dieses Protokolls ist es, eine digitale Kommunikation zu gleichen Bedingungen zu ermöglichen, wie ein vier Augengespräch an einem abhörsicheren Ort.
Beispiel Szenario: Im Folgenden wird ein einfaches Chatbeispiel betrachtet. Der Anwender enthält eine Oberfläche, in der er einen Kommunikationspartner auswählen kann. Nachdem die Auswahl eines Kommunikationspartners abgeschlossen ist, öffnet sich eine neue Maske, welches für jeden Benutzer lediglich aus einem Feld für empfangene Nachrichten und einem Eingabefeld für eigene Nachrichten, sowie einen Knopf um das Geschriebene zu senden, besteht. Zusätzlich ist noch ein Feld enthalten, welches die aktuellen Kommunikationspartner anzeigt. Diese Oberfläche wurde als JavaScript den User Agents von der Webanwendung bereitgestellt. Beim Betätigen der *Senden* Taste werden die gewünschten Crypto-Operationen durchgeführt und die Nachricht an den Kommunikationspartner geschickt.

OTR Merkmale: Das in [14] spezifizierte Protokoll lässt sich in zwei Bereiche aufteilen, die verschiedene Sicherheitsmerkmale voraussetzen. Es wird zwischen den Bereichen *Initialisierung* und *Kommunikation* unterschieden.

Initialisierung: In diesem Bereich authentifizieren sich die beiden Protokollteilnehmer gegenüber dem jeweils anderen. Zusätzlich werden noch weitere Schlüssel ausgetauscht, die für die spätere Kommunikation verwendet werden.

Kommunikation: In diesem Bereich findet die eigentliche Kommunikation zwischen den Protokollteilnehmern statt. Folgende Merkmale sollen dabei erfüllt werden:

- Jede Nachricht ist verschlüsselt um die Vertraulichkeit zu wahren.
- Während der Kommunikation werden keine digitalen Signaturen verwendet, sondern nur MACs. Dieses Vorgehen wird verwendet um die *nicht Zurückweisbarkeit* aufzuheben. Jeder der über den Schlüssel verfügt, mit dem die MACs erstellt wurden, könnte den MAC erzeugt haben. Wenn einer der Protokollteilnehmer jetzt Teile der Kommunikation veröffentlichen sollte, kann er nicht eindeutig beweisen, ob er oder der andere Teilnehmer eine Aussage gemacht hat.
- Für jede Nachricht soll ein eigener Schlüssel verwendet werden, der direkt nach der Verwendung gelöscht wird. So ist der Zugriff auf in der Vergangenheit gesendete Nachrichten nicht möglich, solange sich alle Protokollteilnehmer an diese Bedingung halten.

Vorgehen: OTR ist ein sehr umfangreiches Verfahren, welches den Rahmen dieser Arbeit sprengen würde. Das gesamte Protokoll wird in [14] genau erläutert. An dieser Stelle soll daher nur untersucht werden, ob die einzelnen Operationen über die API zu realisieren sind.

Benötigte Operationen:
Initialisierung: ***[KEYEX-DH]***, ***[MAC]***, ***[RANDOM]***, ***[ENCRYPT]***, ***[DECRYPT]***, ***[SIGN]***, ***[VERIFY]***, ***[DIGEST]***
Kommunikation: ***[ENCRYPT]***, ***[DECRYPT]***, ***[MAC]***, ***[DIGEST]***, ***[VERIFY]***
Alle diese Operationen sind mit der API möglich, vorausgesetzt, dass der User Agent die entsprechenden Algorithmen zur Verfügung stellt.
Probleme: Laut Spezifikation des OTR sollten alle Schlüssel, die nicht mehr benötigt werden, unmittelbar gelöscht werden. Aus diesem Grund dürfen die Schlüssel nicht im Schlüsselspeicher abgelegt werden. Dies passiert allerdings automatisch bei der Erzeugung der Schlüssel. Eine Löschung von Schlüsseln aus dem Speicher ist nicht vorgesehen, da keine entsprechende Methode in [2, Abschn. 14.3] spezifiziert ist.

4.5. Banking

Ziel: Neben der Anmeldung mit Benutzernamen und Passwort soll eine Transaktion mit einer Bank zusätzlich noch durch die Verwendung eines Schlüssels gesichert werden. Zu erwähnen dabei ist, dass der Schlüssel selber keine Rückschlüsse auf den Benutzer zulässt, da er für diese Anwendung extra neu erstellt wird. Dieser Anwendungsfall wird in [13, Abschn. 3.1] eingeführt und beschrieben.
Der dort beschriebene Mechanismus soll verhindern, dass ein Angreifer eine Aktion, wie z.B. eine Überweisung, ausführen kann, wenn er den Benutzernamen und das Passwort eines Benutzers kennt, indem eine Zwei-Faktor Authentifikation durchgeführt wird. Es soll eine Alternative zu *Transaktionsnummern*, kurz *TAN* bieten, die Entweder in Listenform vorliegen oder speziell für jede Transaktion erzeugt werden und vom Benutzer eingegeben werden müssen. Für die Erzeugung von TANs, die speziell für eine Transaktion erzeugt werden, haben sich zwei Verfahren etabliert. Zum einen existiert das *ChipTAN Verfahren*, welches in [15] kurz beschrieben ist und einen Kartenleser und die Bankkarte benötigt, um eine TAN zu erstellen. Weiter existiert das in [15] kurz beschriebene *MobileTAN Verfahren* oder kurz *mTAN* oder auch *smsTAN* genannt. Bei diesem Verfahren wird von der Bank eine SMS an eine hinterlegte Mobilfunknummer gesendet, die die TAN enthält. Der Nachteil an diesen Verfahren ist allerdings, dass der Anwender einen zusätzlichen Schritt durchführen muss, das Eingeben der TAN. Mit dem in [13, Abschn. 3.1] beschriebenen Verfahren soll eine Zwei-Faktor Authentifikation durchgeführt werden, die für den Anwender transparent ist.
Verwendete Funktionalitäten: ***[SIGN]***, ***[DIGEST]***, ***[NAMEDKEY]***, ***[VERIFY]***, ***[KEYCALL]***, ***[ENCRYPT]***, ***[DECRYPT]***, ***[RANDOM]***, ***[UNWRAP]***, ***[WRAP]***, ***[DERIVE-SYM]***, ***[EXPORT]***.
Einige Funktionalitäten werden erst in den nachgelagerten Anwendungsfällen *Protected Document Exchange*, siehe Abschnitt 4.2 und *Document Signing*, siehe Abschnitt 4.3, verwendet.
Voraussetzung: Der Anwender muss bei der Bank ein Konto besitzen. Der Schlüssel wird erst bei der ersten Verwendung des Online-Bankings erstellt. Der Nutzer hat bei Eröffnung des Kontos eine Mobilfunknummer angegeben. Diese Mobilfunknummer wird im Rahmen einer Zwei-Faktor Authentifikation bei der ersten Anmeldung benötigt.
Vorgehen: Der Ablauf ist in 4.3 visualisiert, wobei die Schritte 1 bis 4 nur bei der ersten Anmeldung mit einem neuen User Agent ausgeführt werden. Das Verfahren entspricht dabei der Ausführung einer Aktion,

die durch das *mobileTAN* Verfahren geschützt ist. Für weitere Kommunikation mit der Bank werden diese Schritte Übersprungen.

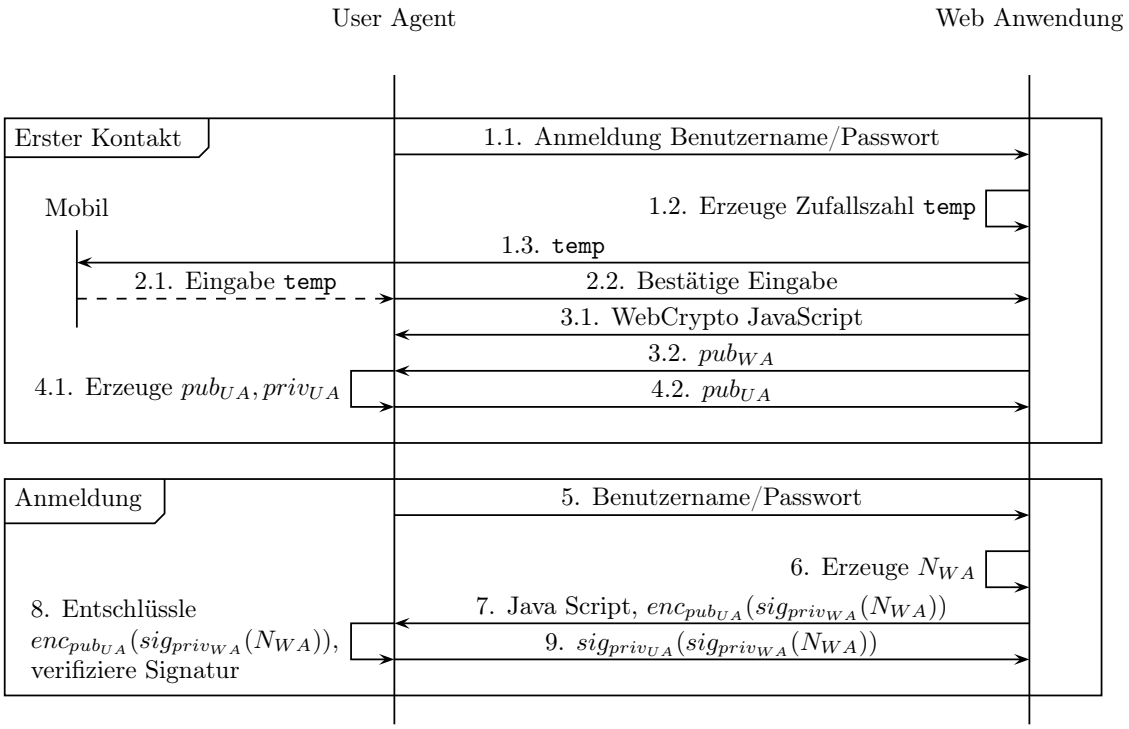


Abbildung 4.3.: Banking, [13, Abschn. 3.1]

1. Der Benutzer meldet sich mit Benutzernamen und Passwort an der Webanwendung an. Diese erzeugt eine Zufallszahl `temp`, welche als Einmalpasswort verwendet wird. `temp` wird an die bei der Kontoeröffnung angegebene Mobilfunknummer gesendet, um sicher zu gehen, dass der Kontoinhaber den Vorgang gestartet hat. Gleichzeitig wird der Benutzer auf eine Seite umgeleitet, auf der er aufgefordert wird `temp` einzugeben. Dieser Schritt ist vergleichbar mit *mTAN* . Darüber hinaus wird er darauf hingewiesen, dass er mit Eingabe des Codes bestätigt, dass es sich nicht um einen öffentlichen Computer handelt und dass er diesen auch in Zukunft nutzen wird. Dies ist notwendig, da in den folgenden Schritten ein Schlüssel erzeugt wird, der im User Agent abgelegt wird. Dieser Schlüssel wird an das Benutzerkonto gebunden und es können nur noch Transaktionen durchgeführt werden, wenn der Schlüssel zur Verfügung steht. Ob es möglich ist, einen anderen User Agent zu nutzen und welche Schritte dazu nötig sind, ist nicht spezifiziert. ***[RANDOM]***

2. Der Benutzer gibt `temp` in die Maske ein.

3. Die Webanwendung lädt den JavaScript Code in den User Agent. Dieser wird benötigt um die wei-

teren Schritte auszuführen. Gleichzeitig übermittelt die Webanwendung ihren öffentlichen Schlüssel pub_{WA}. ***[KEYCALL]***

4. Angestoßen von dem in Schritt 3 übermittelten JavaScript, erzeugt der User Agent nun ein Schlüsselpaar mit dem öffentlichen Schlüssel pub_{UA} und dem privaten Schlüssel $priv_{UA}$. Diese Schlüssel legt der User Agent in seinem Schlüsselspeicher ab. Der eben erzeugte öffentliche Schlüssel pub_{UA} wird an die Webanwendung übermittelt. ***[KEYGEN-ASSYM]***, ***[EXPORT]***

Für die Anmeldung müssen folgende Schritte durchgeführt werden.

5. Der Benutzer ruft die Webanwendung auf und gibt seinen Benutzernamen und Passwort ein.

6. Die Webanwendung erzeugt eine Zufallszahl N_{WA}. Diese wird im laufe des Protokolls verwendet um die Freshness des Protokolllaufes zu sichern. ***[RANDOM]***

7. Es werden die nötigen JavaScript Methoden, sowie das Chiffrat $enc_{pub_{UA}}(sig_{priv_{WA}}(N_{WA}))$, dass die, mit dem öffentlichen Schlüssel pub_{UA} des User Agents, verschlüsselte Signatur über N_{WA} , erstellt mit dem privaten Schlüssel $priv_{WA}$ der Webanwendung enthält, übertragen. ***[SIGN]***, ***[DIGEST]***, ***[ENCRYPT]***, ***[NAMEDKEY]***

8. Die in Schritt 7 übertragene Nachricht $enc_{pub_{UA}}(sig_{priv_{WA}}(N_{WA})), N_{WA}$ wird entschlüsselt und die Signatur $sig_{priv_{WA}}(N_{WA})$ verifiziert. ***[DIGEST]***, ***[VERIFY]***, ***[DECRYPT]***

9. Die in Schritt 8 entschlüsselte Signatur $sig_{priv_{WA}}(N_{WA})$ wird mit dem privaten Schlüssel $priv_{UA}$ des User Agent erneut zu $sig_{priv_{UA}}(sig_{priv_{WA}}(N_{WA}))$ signiert und zusammen mit N_{WA} als $sig_{priv_{UA}}(sig_{priv_{WA}}(N_{WA})), N_{WA}$ an die Webanwendung übertragen. Wenn diese die Signatur verifizieren kann und durch N_{WA} feststellen kann, dass die Nachricht zu dem aktuellen Protokolllauf gehört kann, die Kommunikation gestartet werden. ***[DIGEST]***, ***[VERIFY]***, ***[KEYCALL]***

Das in Schritt 7 übertragene Chiffrat $enc_{pub_{UA}}(sig_{priv_{WA}}(N_{WA})), N_{WA}$ dient dazu, dass beide Parteien sicher sein können, dass die jeweils andere Partei über den erwarteten, privaten Schlüssel verfügt. Durch die Signatur $sig_{priv_{WA}}$ kann der User Agent sicherstellen, dass es sich tatsächlich um die Webanwendung der Bank handelt. Die Verschlüsselung $enc_{pub_{UA}}(sig_{priv_{WA}}(N_{WA}))$ soll dazu verwendet werden, dass die Webanwendung sicher sein kann, dass der User Agent über den passenden privaten Schlüssel verfügt. Dies beweist er ebenfalls mit der in Schritt 9 übertragenden Signatur $sig_{priv_{UA}}(sig_{priv_{WA}}(N_{WA}))$. Im Folgenden können verschlüsselte Dokumente wie in Abschnitt 4.2 beschrieben, ausgetauscht werden oder es können, wie in Abschnitt 4.3 beschrieben, Dokumente signiert werden.

Probleme: Dieses Verfahren birgt dennoch das Risiko, dass ein unerlaubter Zugriff auf das Konto stattfindet, da alle benötigten Daten auf einem Gerät verfügbar sind. Wenn ein Angreifer vollen Zugriff auf den Rechner des Benutzers hat, ist es möglich, dass er ebenfalls Zugriff auf das Schlüsselmaterial erhalten kann. Neben dem Benutzernamen und dem Passwort hat er mit diesem Schlüssel nun alle Informationen, die er benötigt, um z.B. eine Überweisung zu tätigen.

4.6. Authenticated Video Services

Ziel: Dieser, in [13, Abschn. 3.2] eingeführte Anwendungsfall, dient dem sicheren Bereitstellen von *Video on Demand* Diensten, bei denen man, gegen Gebühr, Filme und Serien ansehen kann. In diesem Anwendungsfall werden spezielle Geräte wie Smart-TV und Set-Top Boxen mit Internetzugang betrachtet, auf denen, für den Betrieb auf solchen Geräten, angepasste User Agents verwendet werden. Die klassische Konfiguration mit PC und einem Browser als User Agent wird nicht betrachtet. Da in diesem Anwendungsfall die Kombination von Hardware und einem User Agent betrachtet wird, wird der Begriff *Gerät* verwendet, wenn die Kombination von Hardware und User Agent betrachtet wird.
Neben der reinen Authentifikation des Benutzers gegen den Dienst kann, im Rahmen dieses Anwendungsfalles, auch das Gerät, auf dem sich der User Agent befindet, eindeutig identifiziert werden. Diese Identifizierung dient zu Einen dazu, um zu limitieren, wie viele Geräte ein Benutzer mit seinem Konto verknüpft und zum Anderen, um zu überprüfen, ob das Endgerät die vom Dienstanbieter geforderten Sicherheitsanforderungen einhält. Dabei ist zu beachten, dass „the video in question can only be delivered to devices with certain capabilities that meet the service provider's security requirements, which may vary based on the content and content quality to be delivered“ [13, Abschn. 3.2], also kann der Dienstanbieter verhindern, dass bestimmte Videos auf bestimmten Geräten angesehen werden können, wenn die nicht näher genannten Sicherheitsanforderungen nicht eingehalten werden. Der Grund für dieses Vorgehen ist nicht explizit genannt, soll aber wohl verhindern, dass Inhalte in irgendeiner Form aufgezeichnet oder weiter verbreitet werden.
Dieser Anwendungsfall ist der Ursprung für den in Abschnitt 3.4.2.3.3 beschriebenen *Key Discovery*.
Verwendete Funktionalitäten: ***[KEYGEN-SYM]***, ***[ENCRYPT]***, ***[RANDOM]***, ***[NAMEDKEY]***, ***[DECRYPT]***, ***[EXPORT]***, ***[KEYCALL]***
Voraussetzung: Im entsprechenden Gerät muss ein Identifier und ein Schlüssel K hinterlegt sein. Dieses Paar muss auch dem Service Provider zur Verfügung stehen. Die Bereitstellung der Daten wird vom Hersteller des Gerätes übernommen. Für jeden Service Provider muss zumindest ein eigener Schlüssel hinterlegt werden. Wo dieser Schlüssel im Gerät hinterlegt wird ist nicht spezifiziert, denkbar wäre eine Speicherung im User Agent oder direkt auf einem Modul in der Hardware.
Vorgehen:

1. Bei dem ersten Zugriff auf den Dienst muss der Anwender zunächst ein Benutzerkonto anlegen. Bereits bei der Anmeldung wird der Identifier I_{UA} des User Agent übermittelt.

2. Mithilfe des in Schritt 1 übermittelten Identifiers such die Webanwendung nach dem Schlüssel K, der zu dem Identifiers passt. Weiter überprüft die Webanwendung, ob das Gerät, von dem die Anfrage stammt, die nötigen Sicherheitsanforderungen erfüllt. Der Schlüssel K wird verwendet um zu beweisen, dass das anfragende Gerät nicht nur den Identifier erfahren hat, sondern auch den passenden Schlüssel kennt. Welche Schritte bei der Überprüfung stattfinden wird nicht spezifiziert, es wird lediglich darauf hingewiesen, dass der Schlüssel K so wenig wie möglich verwendet werden soll. ***[NAMEDKEY]***, ***[VERIFY]***, ***[UNWRAP]***, ***[MAC]***, ***[ENCRYPT]***, ***[DECRYPT]***, ***[SIGN]***, ***[DIGEST]***

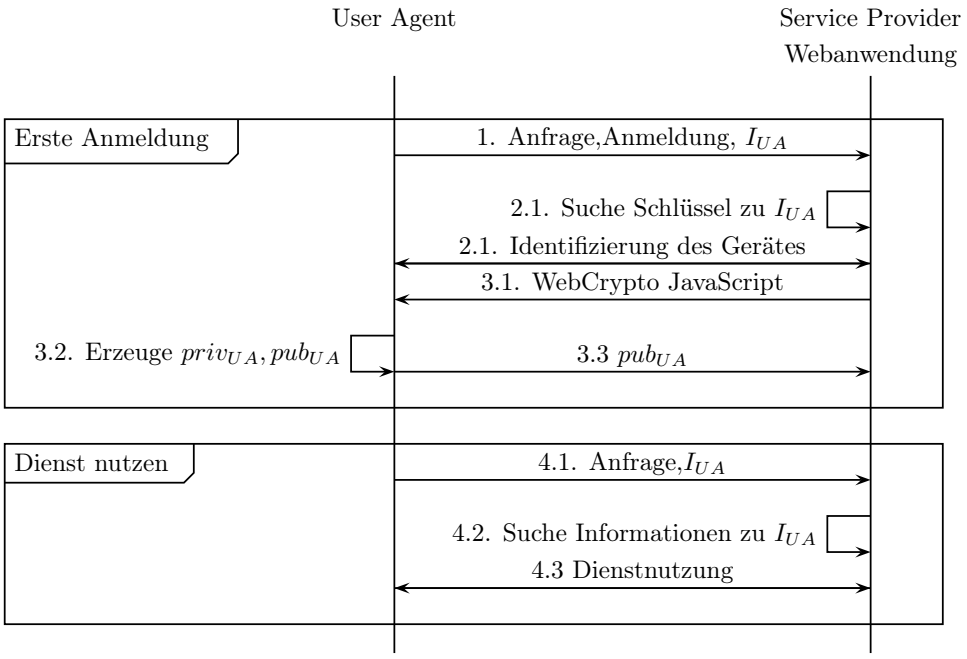


Abbildung 4.4.: Authenticated Video Services, [13, Abschn. 3.2]

3. Im nächsten Schritt erstellt der Benutzer ein Konto für die Webanwendung, dabei erzeugt der User Agent ein Schlüsselpaar und übermittelt pub_{WA} den öffentlichen Teil des Schlüsselpaares an die Webanwendung. Im Rahmen der Kontoerstellung legt der Benutzer ebenfalls fest, wie viele Geräte er mit seinem Konto verwenden möchte. Das Schlüsselpaar kann nun verwendet werden um den Benutzer eindeutig zu identifizieren. Die Parameter, die für die Erstellung des Schlüsselpaares verwendet werden sind nicht spezifiziert, da als Funktion jedoch ***[DERIVE-ASSYM]*** angegeben ist und nicht ***[KEYGEN-ASSYM]***, kann davon ausgegangen werden, dass der Schlüssel K als Eingangswert für die Erstellung des Schlüsselpaares verwendet wird. ***[DERIVE-ASSYM]***, ***[EXPORT]***, ***[KEYCALL]***, ***[SIGN]***, ***[VERIFY]***, ***[DIGEST]***, ***[KEYEX]***

4. Bei der Nutzung des Dienstes wird zunächst eine Anfrage an die Webanwendung erstellt. Durch die Übermittlung des Wertes I_{UA} kann diese nun den Benutzer identifizieren und die entsprechenden Schlüssel bereit stellen. Ebenfalls kann die Webanwendung feststellen, ob das anfragende Gerät die nötigen Sicherheitsanforderungen erfüllt.
   Der weitere Ablauf ist nicht näher spezifiziert. Er wird lediglich erwähnt, dass ein sicherer Kanal verwendet wird und dass der Anbieter anhand der Sicherheitsanforderungen feststellt, ob ein Video wiedergegeben wird. Durch die spezifizierten verwendeten Methoden wäre eine Authentifizierung des Benutzers anhand seines öffentlichen Schlüssels möglich. Das Video selber scheint durch das Fehlen der Verwendung von ***[DECRYPT]*** nicht verschlüsselt zu sein. ***[NAMEDKEY]***, ***[KEYCALL]***,

[SIGN], ***[DIGEST]***, ***[VERIFY]***, ***[MAC]***, ***[WRAP]***, ***[ENCRYPT]***

Dieser Anwendungsfall ist sehr undeutlich formuliert und es belieben einige Fragen offen, wie z.B. eine Anbindung weiterer Geräte funktioniert. Durch die Formulierung „[the] account creation involved the creation of a specific key pair to uniquely identify [the user], and safely exchanges keys with the video service's servers“ [13, Abschn. 3.2], lässt darauf schließen, dass dieses Schlüsselpaar auf jedem Gerät, das verwendet werden soll, zur Verfügung stehen muss.

4.7. Code Sanctity and Bandwidth Saver

Ziel: Eine Webanwendung möchte beliebigen JavaScript Code in einer Bibliothek auf dem User Agent ablegen, damit die Daten nicht bei jedem Aufruf der Webanwendung übertragen werden müssen. Bei einer Änderung in dieser Bibliothek soll die alte Version, die im User Agent gespeichert ist, durch die neue Version ersetzt werden. Dieser Anwendungsfall wird in [13, Abschn. 3.3] beschrieben.
Verwendete Funktionalitäten: ***[DIGEST]***
Voraussetzung: Für diesen Anwendungsfall existieren keine besonderen Voraussetzungen.
Vorgehen:

1. Der User Agent ruft die Webanwendung auf, die den benötigten WebCrypto JavaScript Code sowie den Hash-Wert der aktuellen Version der Bibliothek an den User Agent übermittelt. ***[DIGEST]***
2. Im nächsten Schritt erzeugt der User Agent einen Hash-Wert über seine Kopie der Bibliothek und vergleicht diese mit dem von der Webanwendung übermittelten Wert. ***[DIGEST]***
3. Wenn beide Werte übereinstimmen, wird die lokale Kopie verwendet. Wenn keine lokale Kopie vorhanden ist oder der Hash-Wert nicht mit dem Übermittelten übereinstimmt, wird die neue Version der Bibliothek von der Webanwendung übermittelt.

Bei Bedarf kann man den von der Webanwendung übermittelten Hash-Wert zusätzlich durch eine Signatur schützen, die vom User Agent vor dem Herunterladen einer neuen Version überprüft wird. ***[SIGN]***, ***[VERIFY]***, ***[DIGEST]***

4.8. Verschlüsselte Kommunikation via Webmail

Ziel: Es soll möglich sein, dass ein Anwender eine verschlüsselte E-Mail über ein Webmail-Interface versenden können soll. Bisher ist dafür ein Mail Client nötig. Dieser Anwendungsfall wird in [13, Abschn. 3.4] beschrieben.
Verwendete Funktionalitäten: ***[DIGEST]***, ***[SIGN]***, ***[NAMEDKEY]***, ***[VERIFY]***, ***[ENCRYPT]***, ***[DECRYPT]***, ***[IMPORT]***, ***[WRAP]***, ***[UNWRAP]***, ***[KEYGEN-SYM]***
Voraussetzung: Jede Partei verfügt bereits über ein Schlüsselpaar und stellt seinen öffentlichen Schlüssel allgemein zugänglich zur Verfügung.

Vorgehen: In diesem Beispiel, wie in Abb. 4.5 zu sehen ist, wird von einer Kommunikation zwischen zwei Parteien ausgegangen, die zu dem ersten Mal miteinander kommunizieren. Dieses Verfahren lässt sich allerdings einfach auf eine Kommunikation von mehr als zwei Parteien ausweiten. Des Weiteren wird, aus gründen der Übersichtlichkeit, angenommen, dass beide Parteien denselben Anbieter verwenden und die Webanwendung auf demselben Host wie der Mail-Server betrieben wird.

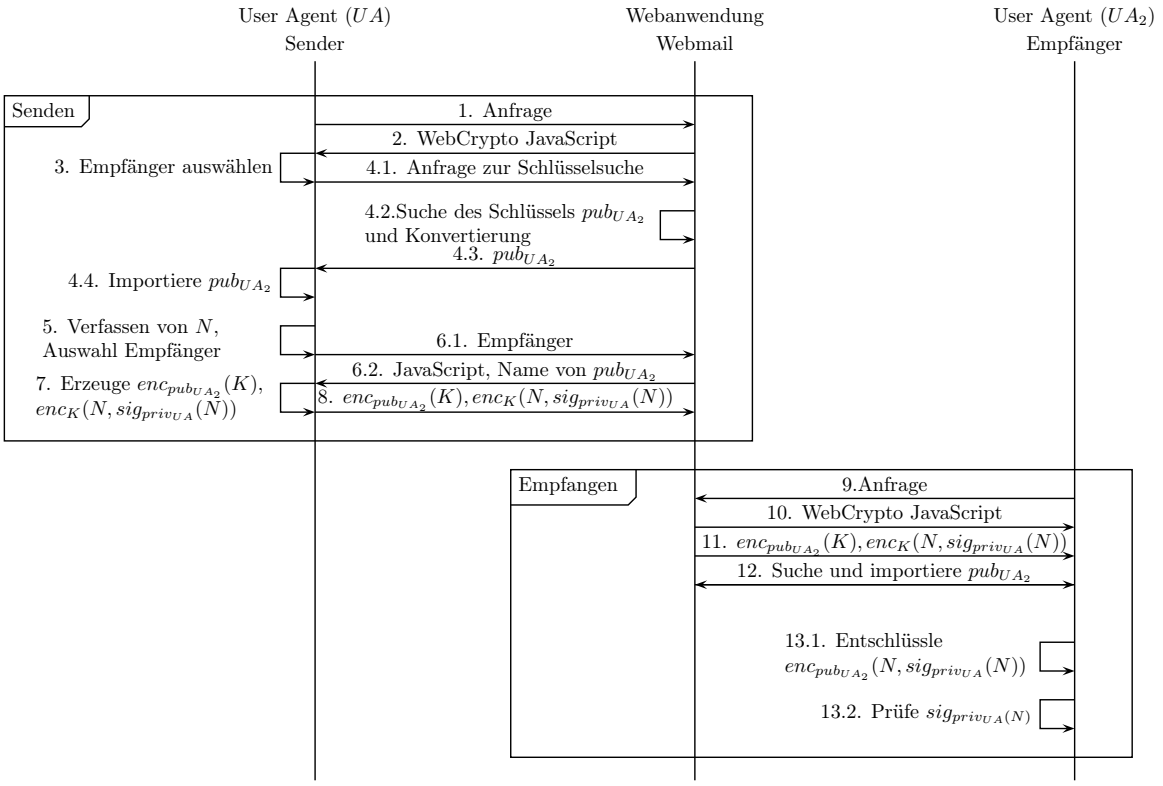


Abbildung 4.5.: Verschlüsselte Kommunikation via Webmail, [13, Abschn. 3.4]

1. Der User Agent stellt die Anfrage an die Webanwendung. In diesem Fall bedeutet das, dass der Benutzer eine Website aufruft, die ihm eine gewohnte Oberfläche zu dem verfassen einer E-Mail, sowie zum importieren von Schlüsseln bereitstellt.

2. Das benötigte WebCrypto Java Script wird an den User Agent übermittelt.

3. Es wird ein Kommunikationspartner ausgewählt.

4. Der Anwender gibt an, welcher Schlüssel für die Kommunikation verwendet werden soll. Dazu gibt er die Quelle an, von der der Schlüssel pub_{UA_2} importiert werden soll. Die Webanwendung sucht nun diesen Schlüssel, konvertiert ihn in ein Format, das die WebCrypto API verarbeiten kann und importiert den Schlüssel in den Schlüsselspeicher des User Agents. Eine Auflistung über unterstützte

Formate gibt Tabelle 3.5. Um diesen Schlüssel später nutzen zu können, muss erkennbar sein, zu welchem Empfänger dieser Schlüssel gehört. Wie diese Zuordnung stattfinden soll, ist nicht spezifiziert. Eine Möglichkeit wäre es, der Webanwendung mitzuteilen, für welchen Kommunikationspartner der Schlüssel, der importiert werden soll, zu verwenden ist. Dies könnte z.B. durch die E-Mail Adresse des Schlüsselinhabers geschehen. ***[IMPORT]***

5. Nachdem der Schlüssel importiert worden ist kann eine sichere Kommunikation stattfinden. Dazu verfasst der Absender wie gewohnt eine Nachricht N und wählt den Empfänger aus. Für diesen Empfänger muss bereits ein Schlüssel importiert worden sein.

6. Beim Absenden der Mail wird zunächst der gewünschte Empfänger an die Webanwendung übermittelt. Diese sendet das benötigte JavaScript sowie den Namen des öffentlichen Schlüssels des Empfängers zurück an den User Agent.

7. Der User Agent signiert nun N mit seinem privaten Schlüssel $priv_{UA_1}$ die Signatur $sig_{priv_{UA_1}}(N)$. Daraufhin erzeugt er einen symmetrischen Schlüssel K, womit er die Signatur der Nachricht N zusammen mit N zu $enc_K(sig_{priv_{UA_1}}(N), N)$ verschlüsselt. Zusätzlich wird der Schlüssel K mit dem öffentlichen Schlüssel pub_{UA_2} des Empfängers verschlüsselt . ***[NAMEDKEY]*** ***[DIGEST]***, ***[SIGN]***, ***[ENCRYPT]***, ***[WRAP]***, ***[KEYGEN-SYM]***, ***[KEYCALL]***

8. Das grade erstellte Chiffrat $enc_{pub_{UA_2}}(K), enc_K(N, sig_{privUA}(N))$ wird nun an den Webserver übermittelt und von dort aus an den Anbieter des Empfängers weitergeleitet.

9. Der Kommunikationspartner ruft die Webanwendung auf, die wiederum einen Webmail Client darstellt, der die bekannten Funktionalitäten bereit stellt.

10. Die Webanwendung übermittelt das benötigte WebCrypto JavaScript zusammen mit der Webmailoberfläche an den User Agent. Der Anwender bekommt die Mitteilung, dass eine neue Nachricht verfügbar ist.

11. Das Chiffrat wird an den Empfänger übermittelt.

12. Da noch kein öffentlicher Schlüssel des Absenders registriert ist, wird der Empfänger aufgefordert diesen zu importieren. Wie diese Erkennung stattfindet ist nicht spezifiziert, denkbar wäre jedoch eine Fehlermeldung bei der Schlüsselsuche auszuwerten. Das Importieren des Schlüssels würde, wie in Schritt 4 beschrieben, stattfinden. ***[IMPORT]***

13. Nun wird der Schlüssel K entschlüsselt, danach die Nachricht N und anschließend wird die Signatur überprüft. ***[NAMEDKEY]***, ***[DIGEST]***, ***[VERIFY]***, ***[DECRYPT]***, ***[UNWRAP]***

4.9. BrowserID

4.9.1. *assertionPlusCert*

Im Single Sign-On Umfeld werden $assertion$ verwendet, um dem Service Provider zu bestätigen, dass sich ein Benutzer gegenüber einem Identity Provider authentifiziert hat. Zusätzlich beinhaltet diese $assertion$ Informationen zu der angefragten Ressource und zum Gültigkeitszeitraum. Das Problem mit diesen $assertion$ ist, dass sie durch Angriffe wie *XSS* oder *Man-in-the-Middle* gestohlen und vom Angreifer verwendet werden können. Eine Übersicht über mögliche Angriffe gibt [16] am Beispiel eines SSO-Verfahrens unter Verwendung von *SAML*, der *Security Assertion Markup Language*. Nähere Informationen zu SAML gibt [17]. In diesem, in [13, Abschn. 3.5] spezifizierten, Anwendungsfall wird die $assertion$ zu einer $assertionPlusCert$ erweitert. Ein Beispiel gibt Listing 4.1.

```
// This is for illustrative purposes only
// Proper use of JWT uses Base64
assertionPlusCert =
{
  "assertion": {
    "audience": "webapplication.example",
    "valid-until": 1308859352261,
  }, // signed using users's private key
       // minted by Identity Provider for User@mail.example
  "certificate": {
      "email": "User@mail.example",
      "public-key": "",
      "valid-until": 1308860561861,
  } // certificate is signed by Identity Provider
};
```

Listing 4.1: UseCases]assertionPlusCert, Quelle: [13, Abschn. 3.5]

Eine $assertionPlusCert$ beinhaltet zum Einen eine $assertion$ und zum Anderen ein Zertifikat $cert$. In $cert$ signiert der Identity Provider den öffentlichen Schlüssel des Anwenders zusammen mit Informationen zu dem Benutzer und der Gültigkeitsdauer des Zertifikates. Wenn sich nun ein Benutzer beim Identity Provider erfolgreich authentifiziert, erstellt der Identity Provider die $assertion$, die für die Anmeldung an den Dienst benötigt wird. Diese $assertion$ wird nun an den User Agent des Benutzers übermittelt, welcher die $assertion$ mit seinem eigenen privaten Schlüssel signiert. Der private Schlüssel, mit dem die Signatur erzeugt wird, muss zu dem öffentlichen Schlüssel passen, der in dem Zertifikat $cert$ beinhaltet ist. Der User Agent übermittelt die signierte $assertion$ zusammen mit dem Zertifikat $cert$ als $assertionPlusCert$ an den Service Provider. Dieser prüft, ob die Signatur des Zertifikates gültig ist. Wenn die Signatur gültig ist, prüft er mit dem öffentlichen Schlüssel des Benutzers aus dem Zertifikat $cert$, ob die Signatur über die

$assertion$ gültig ist. Wenn dies der Fall ist, dann wird der Zugriff auf die angefragte Ressource gewährt. Dies soll verhindern, dass die $assertion$ gestohlen und verwendet wird, da ein Angreifer nicht in der Lage ist, die Signatur über die $assertion$ zu erzeugen. Wie die Überprüfung der Signaturen umgesetzt wird, ist nicht beschrieben.

4.9.2. Verfahren

BrowserID ist ein Protokoll für Single Sign-On, welches sich mit Hilfe der Methoden der WebCrypto API implementieren lässt. Beschrieben wird dieser Anwendungsfall in [18]. Ein beispielhafter Durchlauf des Protokolls ist in Abb. 4.6 gegeben. Im Rahmen dieses Anwendungsfalls wird eine Assertion als Anmeldetoken im ***[JWT]***, *JSON Web Token* Format, welches in [19] beschrieben wird, erzeugt. Eine beispielhafte Darstellung ist im Listing 4.1 zu sehen. Im Diagramm selber sind die einzelnen kryptografischen Operationen, anders als bei den anderen Anwendungsfällen nicht explizit aufgeführt, da diese in Erstellung der Assertion $assertion$ gekapselt sind. In der folgenden Beschreibung des Ablaufes wird auf die einzelnen Operationen bei der Erstellung der $assertionPlusCert$ eingegangen. Eingeführt wird dieser Anwendungsfall in [13, Abschn. 3.5].

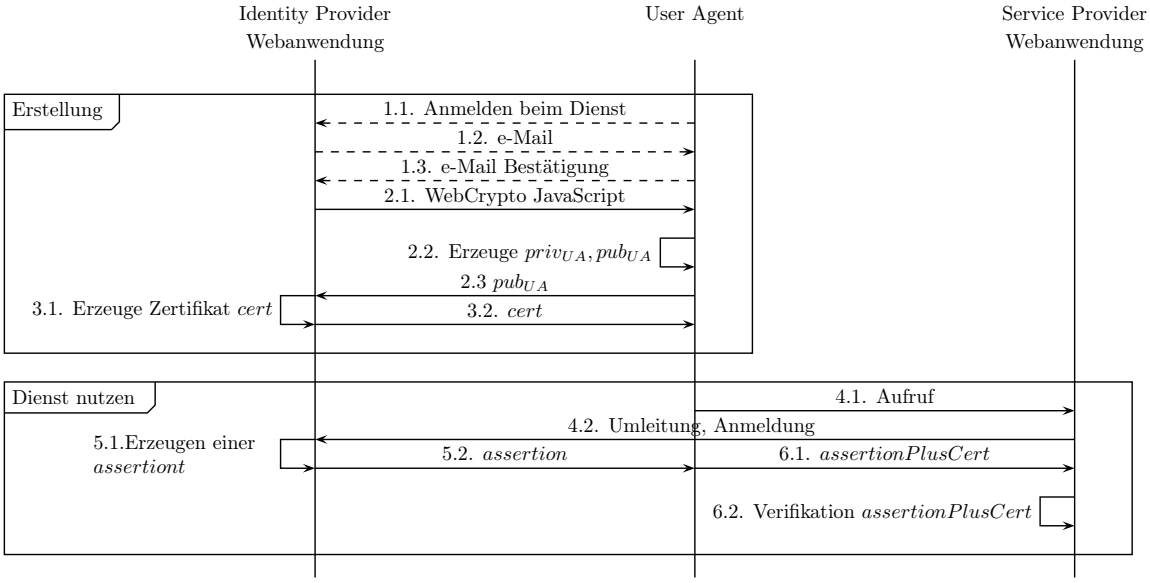


Abbildung 4.6.: BrowserID, [13, Abschn. 3.5]

Ziel: Anmeldung an eine Webanwendung eines Service Providers mit Hilfe eines Identity Providers und Sigle-Sign On. Bei Single-Sign On wird nur noch ein Benutzername/Passwort für mehrere Webanwendungen verwendet. Die Anmeldung erfolgt bei einem Identity Provider, der die Authentifikation gegenüber angeschlossenen Service Providern, der die angefragte Ressource bereitstellt, übernimmt. Dazu wird eine,

in 4.9.1 beschriebene, $assertionPlusCert$ ausgestellt, welche ein erweitertes Anmeldetoken ist, dass mit Hilfe eines Zertifikates, an einen bestimmten User Agent gebunden wird. Ein weiteres Ziel ist es, dass z.B. durch einen *Man-in-the-Middle* Angriff gestohlene Tokens nicht verwendet werden können.
Voraussetzungen: Die Webanwendung des Service Providers muss mit dem Identity Provider zusammen arbeiten.
Verwendete Funktionalitäten: ***[KEYGEN-ASSYM]***, ***[EXPORT]***, ***[KEYCALL]***, ***[SIGN]***, ***[VERIFY]***, ***[IMPORT]***, ***[DIGEST]***
Vorgehen:

1. Der erste Schritt beschreibt die Registrierung bei dem Identity Provider. Im Rahmen dieser Anmeldung werden keine kryptografischen Operationen benötigt und kann sich von Anbieter zu Anbieter unterscheiden. Aus diesem Grund ist die Kommunikation gestrichelt dargestellt.
   Zunächst beginnt der Anwender den Registrierungsvorgang bei dem Identity Provider. Dazu hinterlegt er eine E-Mail Adresse, in diesem Beispiel *User@mail.example*. Daraufhin sendet der Identity Provider einen Registrierungslink an die hinterlegte Adresse. Nachdem der Anwender mit Klick auf diesen Link bestätigt hat, dass er Besitzer dieser Adresse ist, startet der eigentliche Registrierungsvorgang.

2. Mit Klick auf den Link wird der Anwender auf eine Webanwendung des Identity Providers umgeleitet, der JavaScript Methoden der API übermittelt, die den User Agent veranlassen, ein Schlüsselpaar $priv_{UA}, pub_{UA}$ zu erstellen und den öffentlichen Teil an den Identity Provider zurückzusenden. ***[KEYGEN-ASSYM]***, ***[EXPORT]***

3. Der Identity Provider erzeugt nun ein Zertifikat $cert$ für den Anwender. In diesem Zertifikat ist der öffentliche Schlüssel pub_{UA} des Anwenders, die hinterlegte E-Mail Adresse und ein Gültigkeitszeitraum enthalten. Dieses Zertifikat signiert der Identity Provider mit seinem privaten Schlüssel. Das Zertifikat wird an den User Agent zurück übermittelt und dort gespeichert. ***[KEYCALL]***, ***[SIGN]***, ***[DIGEST]***

Die Registrierung bei dem Identity Provider ist nun abgeschlossen. Nun können diese Informationen verwendet werden, um sich bei einer angeschlossenen Webanwendung anzumelden.

4. Bei dem Aufruf der Webanwendung wird nun als Identity Provider der Anbieter ausgewählt, bei dem die Registrierung durchgeführt wurde. Der Anwender wird, für ihn transparent, zu dem Identity Provider umgeleitet. Wie sich der Anwender gegenüber dem Identity Provider authentifiziert wird spezifiziert.

5. Der Identity Provider erzeugt nun die eigentliche *assertion*, die Informationen über den Service Provider und die Gültigkeit enthält. Die $assertion$ wird nun zu dem User Agent übermittelt, welcher sie mit dem privaten Schlüssel pub_{UA} vom Anwender signiert. ***[SIGN]***, ***[DIGEST]***, ***[NAMEDKEY]***

6. Der User Agent fügt die $assertion$ mit $cert$ zu $assertionPlusCert$ zusammen und leitet $assertionPlusCert$ im nächsten Schritt an die Webanwendung weiter. Dort werden nun zunächst

der öffentliche Schlüssel pub_{IP} des Identity Providers sowie der öffentliche Schlüssel pub_{UA} des Benutzers gesucht. Wenn beide Schlüssel vorliegen prüft die Webanwendung die Signatur über den Zertifikatsteil von $assertionPlusCert$, in dem der öffentliche Schlüssel vom Anwender noch einmal hinterlegt ist. Wenn diese Signatur erfolgreich validiert wurde, vergleicht die Webanwendung den gefunden und den im Zertifikat signierten öffentlichen Schlüssel pub_{UA} vom Anwender miteinander. Stimmen diese überein, überprüft die Webanwendung die Signatur über den *Assertion* Teil von $assertionPlusCert$. Wenn diese erfolgreich validiert wurde, wird der Anwender angemeldet. **`[KEYCALL]`**, **`[VERIFY]`**, **`[IMPORT]`**

Vorstellbar wäre auch eine Adaption diese Vorgehens für andere Single Sign-On Verfahren. Dafür müsste das $assertionPlusCert$ Konstrukt für das entsprechende Verfahren angepasst werden. So könnte beispielsweise eine *SAML-Assertion* in dieses Konstrukt eingebettet werden, welche dann vom User Agent signiert wird.

5. Weitere Anwendungsfälle

In diesem Kapitel werden weitere Anwendungsfälle für die WebCrypto API besprochen, die nicht in den Dokumenten des Standards erwähnt wurden. Im Übrigen gelten für alle Anwendungsfälle die gleichen Annahmen und Konventionen wie in Kapitel 4.
Die hier beschriebenen Anwendungsfälle, mit Ausnahme von 5.3 *Soft Token*, bauen aufeinander auf. So werden für den Anwendungsfall 5.2 Zertifikate verwendet, die wie in 5.1 erstellt wurden. Im Anwendungsfall 5.2.1.2.1, können wiederum Aspekte aus dem Anwendungsfall 5.2 verwendet werden.

5.1. Erzeugen eines Client Zertifikates

5.1.1. Motivation

Mit einem Client Zertifikat kann man eine, in [20] spezifizierte, *mutual authenticated TLS Session* erzeugen, bei der sich zusätzlich der User Agent gegenüber der Webanwendung authentifiziert. Eine solche Authentifikation könnte auch als eine Art Visitenkarte verwendet werden, mit der der User Agent der Webanwendung alle Informationen bereitstellt, die sie benötigt, um den passenden WebCrypto API-Schlüssel für weitere kryptografische Operationen zu finden. So könnte die Webanwendung, ohne weitere Informationen zu benötigen, einen *Key Discovery* Vorgang, wie in 3.4.2.3.3 beschrieben, anstoßen. Diese Zertifikate könnten dynamisch erzeugt werden, der Anwender müsste also nicht wie bisher, im Vorfeld ein Zertifikat importieren.
Das Vorgehen für die Auswahl des passenden Zertifikates hängt dabei vom Browser ab. Eine Lösung wäre es, dass der Anwender selber aus einer Liste zur Verfügung stehender Zertifikate wählen kann, was nach [21], zur Zeit Standard ist. Diese Möglichkeit würde allerdings wieder eine Sicherheitsentscheidung vom Anwender vorauszusetzen, was in der Regel kritisch ist, wie [6] zeigt. Eine weitere Möglichkeit wäre es, das passende Zertifikat über den aufgerufenen Host zu ermitteln. Für diese Möglichkeit wäre eine Anpassung des Browsers notwendig. Ein entsprechendes Verfahren wird in Abschnitt 5.2 erläutert.
Darüber hinaus können Client Zertifikate auch verwendet werden um *secure Bindings*, ein Verfahren um SSO-Authorisierungstokens mit Informationen aus einer bestimmten TLS-Verbindung zu verknüpfen, umzusetzen. Ein Verfahren, das *Holder of Key* Verfahren, welches in [22] eingeführt wird und das Token an ein Client Zertifikat bindet, wird in Abschnitt 5.2.1.2.1 dargestellt.

5.1.2. Vorgehen

Ziel: Es soll ein Client-Zertifikat erzeugt werden, das für spätere TLS/SSL Sitzungen mit der Webanwendung genutzt werden soll.
Voraussetzungen: Der Speicher für Client Zertifikate sowie die TLS-Engine des Browsers müssen auf den Schlüsselspeicher der WebCrypto-API zugreifen können, um den öffentlichen Schlüssel des Zertifikates mit dem privaten Schlüssel, der API-Schlüsselspeicher hinterlegt ist, verbinden zu können.
Verwendete Funktionalitäten: ***[KEYGEN-ASSYM]***, ***[EXPORT]***, ***[KEYCALL]***, ***[DIGEST]***, ***[SIGN]***
Vorgehen: In diesem Beispiel erwartet ein Webserver die Verwendung eines *TLS Client Zertifkates*. Der Anwender wird nach einem Zertifikat gefragt, wenn er keines angibt, wird er auf eine Webanwendung umgeleitet, die ein Zertifikat erstellt.
Der Ablauf zum Erzeugen eines Client Zertifikates ist in Abb. 5.1 dargestellt.

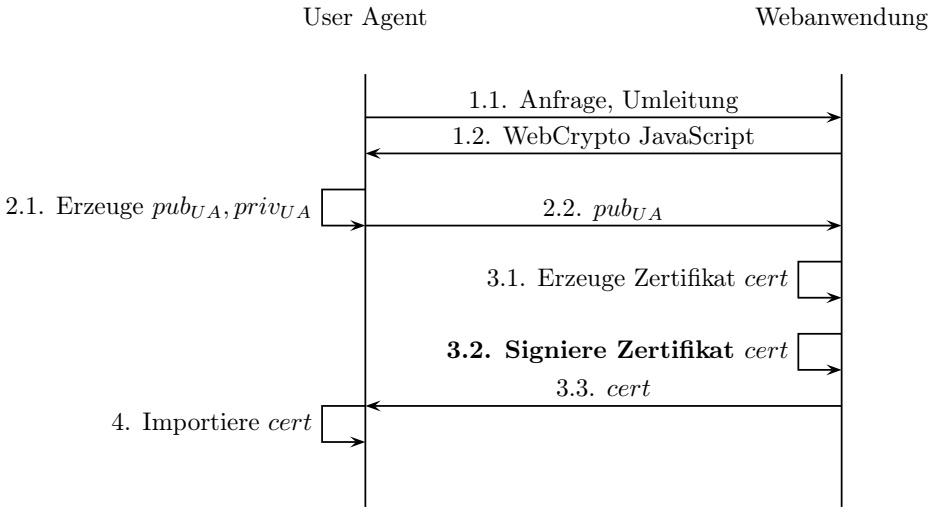


Abbildung 5.1.: Client Zertifikat Erstellung - Von der Webanwendung signiert

1. Der User Agent stellt die erste Anfrage an den Webserver. Dieser stellt fest, dass der User Agent kein Zertifikat vorweisen kann. Daraufhin leitet er den User Agent auf eine Webanwendung um, die die nötigen JavaScript Methoden für eine Zertifikatserstellung überträgt und den Erstellungsvorgang einleitet.

2. Im nächsten Schritt erstellt der User Agent ein Schlüsselpaar und sendet den öffentlichen Teil des Schlüssels an die Webanwendung. ***[KEYGEN-ASSYM]***, ***[EXPORT]***

3. Mit dem öffentlichen Schlüssel des User Agent erzeugt die Webanwendung nun ein Zertifikat $cert$, signiert dieses mit ihrem eigenen privaten Schlüssel und sendet das Zertifikat anschließend an den

User Agent zurück. ***[KEYCALL]***, ***[SIGN]***, ***[DIGEST]***

4. Der User Agent importiert nun dieses Zertifikat in seinen Zertifikatsspeicher und teilt dem Anwender mit, welches Zertifikat er für die Verbindung zu der Webanwendung verwenden soll.

Dieses Vorgehen erzeugt ein, von der Webanwendung signiertes, Zertifikat. Ebenfalls ist es möglich, dass der User Agent das Zertifikat selber signiert. Solche selbst signierten Zertifikate werden für die, in [21] beschriebene, Erzeugung von *Origin-Bound Certificates* benötigt. Der oben beschriebene Vorgang muss dazu leicht abgewandelt werden. Die Unterschiede sind in den Grafiken hervorgehoben. Das Verfahren wird in Abb. 5.2 dargestellt:

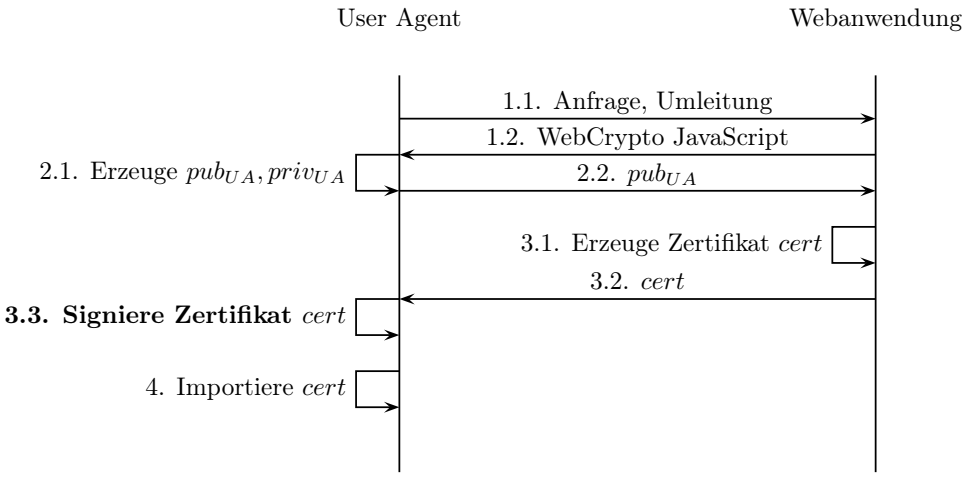


Abbildung 5.2.: Client Zertifikat Erstellung - Vom User Agent signiert

1. Der User Agent stellt die erste Anfrage an den Webserver, dieser stellt fest, dass der User Agent kein Zertifikat vorweisen kann. Daraufhin leitet er den User Agent auf eine Webanwendung um, die die nötigen JavaScript Methoden für eine Zertifikatserstellung überträgt und den Erstellungsvorgang einleitet. Dieser Schritt ist identisch zu dem bei der Erstellung von Zertifikaten, die von der Webanwendung signiert werden.

2. Im nächsten Schritt erstellt der User Agent ein Schlüsselpaar und sendet den öffentlichen Teil des Schlüssels an die Webanwendung. ***[KEYGEN-ASSYM]***

3. Mit dem öffentlichen Schlüssel des User Agent erzeugt die Webanwendung nun ein Zertifikat $cert$ und sendet das Zertifikat anschließend an den User Agent zurück. Dieser signiert das Zertifikat nun mit seinem privaten Schlüssel. ***[KEYCALL]***, ***[SIGN]***, ***[DIGEST]***

4. Der User Agent importiert nun dieses Zertifikat in seinen Zertifikatsspeicher und teilt dem Anwender mit, welches Zertifikat er für die Verbindung zu der Webanwendung verwenden soll.

Ähnliche Funktionalitäten bietet auch das *keygen-Tag* von HTML5, welches in [23] spezifiziert wird. Das *keygen-Tag* stößt auf dem Client die Erzeugung eines Schlüsselpaares an. Der private Teil wird direkt im Schlüsselspeicher des Clients gespeichert, der öffentliche Teil wird an den Server zurück gesendet.

5.2. Origin-Bound Certificates

Origin-Bound Certificates wurden 2012 in [21] als Erweiterung für etablierte Authentifikationsmethoden, im speziellen Benutzername/Passwort und Cookies, vorgeschlagen. Die Idee hinter dem Verfahren besteht darin, für jede *origin*, also für jede Webanwendung die man verwenden möchte, ein TLS-OBC (TLS *Origin-Bound Certificate*) erstellt wird. Der Benutzer muss sich anschließend nur noch einmal mit seinem Benutzernamen und Passwort an eine Webanwendung anmelden. Mit Hilfe von *Origin-Bound Certificates* wird anschließend ein *Channel-binding cookie*, wie in Abschnitt 5.2.1.1 beschrieben, erstellt. Dieses wird im Folgenden dann zur Authentifikation des Nutzers verwendet.
Dabei ist ein*Origin-Bound Certificate* vergleichbar mit einem TLS-Client Zertifikat, mit dem Unterschied, dass keine Nutzerinteraktion vorgesehen ist. Der Browser sucht selbstständig nach dem passenden Zertifikat für die entsprechende Webanwendung. Wenn keines vorhanden ist, wird ein neues Zertifikat erstellt. Die Erstellung ist für den Nutzer transparent. Bisher war es notwendig, dass der Anwender ein Client Zertifikat manuell auswählen muss [21]. Nach Aussage der Autoren in [21, Abschn. 3.2], können die so erzeugten Zertifikate, wie herkömmliche TLS-Client Zertifikate, in einem TLS Handshake verwendet werden, ohne das dieser verändert werden muss. Da die Erstellung der Zertifikate automatisch geschieht und keine Nutzerinformationen benötigt, kann keine Identifizierung eines Nutzers allein anhand des Zertifikates erfolgen. Die Zertifikate werden dabei vom User Agent signiert.
Die Erstellung eines solchen Zertifikates ist im Anwendungsfall *Erstellung eines Client Zertifikates* in Abschnitt 5.1 beschrieben und kann ohne Änderungen für die Erzeugung von *Origin-Bound Certificates* verwendet werden.
Origin-Bound Certificates benötigen Anpassungen im Browser, um Zertifikate erstellen zu können und um diese automatisch der richtigen Webanwendung zuordnen zu können. Die Erstellung kann durch Methoden der WebCrypto API erfolgen. Da die Auswahl der Zertifikate allerdings bereits auf TLS Ebene geschieht, also bevor die API für die Webanwendung zur Verfügung steht, kann die Auswahl nicht durch die API durchgeführt werden. Ohne Anpassung des Browsers müsste der Anwender die Zertifikate selber zuordnen und damit Sicherheitsentscheidungen treffen.

5.2.1. Anwendungsmöglichkeiten

5.2.1.1. Channel-binding cookies

Eine Anwendungsmöglichkeit für diese Zertifikate stellt das Binden von Cookies an eine bestimmte TLS Verbindung da und wird in [21] vorgeschlagen. Ziel soll es sein, dass Cookies nur noch in Verbindung mit einem speziellen Zertifikat verwendet werden können. Auf diese Art und Weise soll verhindert werden, dass z.B. durch XSS-Angriffe gestohlene Cookies zur Anmeldung an eine Webanwendung genutzt werden. Eine

in [21] vorgeschlagene Struktur der Cookies ist:

$$cookie =< v, HMAC_K(v + f) >$$

Dabei stellt v einen herkömmlichen Cookie da, der beispielsweise Anmeldeinformationen enthält. f ist der Fingerprint des öffentlichen Schlüssels aus dem *Origin-Bound Certificate*, $(v + f)$ ist die einfache konkatinierung der Werte v und f. $HMAC$ beschreibt die Erstellung eines *MAC* über den Hash-Wert von $v + f$ unter Verwendung des Schlüssels K, der nur der Webanwendung bekannt ist.
Ein beispielhafter Ablauf könnte wie folgt aussehen:
Vorausetzungen: Der Client verfügt über ein Origin-Bound Certificate zur Kommunikation mit dem Server der Webanwendung.

1. Der User Agent baut eine Verbindung mit der Webanwendung auf. Es wird ein *mutual authenticated TLS Kanal* mit Hilfe des Origin-Bound Certificate des Client erstellt.

2. Die Webanwendung möchte ein Cookie setzen. Dazu erzeugt er eines in der oben angegebenen Form. Der Beweis, dass des User Agent im Besitz des zu dem Origin-Bound Certificate gehörenden privaten Schlüssels ist, findet bereits während des TLS-Handshake statt.

3. Das Cookie $cookie$ wird an den User Agent übertragen und dort gespeichert.

Zu dem jetzigen Zeitpunkt ist $cookie$ erstellt und im User Agent des Anwenders abgelegt. Wenn das Cookie nun verwendet werden soll, müssen folgende Schritte ausgeführt werden.

4. Der User Agent baut eine Verbindung mit der Webanwendung auf. Es wird erneut ein *mutual authenticated TLS Kanal* mit Hilfe des Origin-Bound Certificate des Client erstellt. Das hier verwendete Origin-Bound Certificate muss dasselbe sein, wie bei der Erstellung von $cookie$.

5. Nun überträgt der User Agent das Cookie $cookie$ zur Webanwendung, diese erstellt erneut den $HMAC_K(v + f)$ und überprüft ihn mit dem in $cookie$ übermittelten Wert. Wenn beim Verbindungsaufbau ein anderes Origin-Bound Certificate verwendet wurde als bei der Erstellung von $cookie$, kann das die Webanwendung nun erkennen und das Cookie verwerfen. So müsste ein Angreifer, neben $cookie$ auch noch das Origin-Bound Certificate des Client besitzen um $cookie$ nutzen zu können. Da der HMAC mit dem Schlüssel K erstellt wird, den nur die Webanwendung kennt, kann ein Angreifer diesen auch nicht neu erzeugen, um ihn seinem Origin-Bound Certificate anzupassen.

Bis auf die Zertifikatserstellung werden alle kryptografischen Operationen von der Webanwendung durchgeführt.

5.2.1.2. Single Sign-On

5.2.1.2.1. TLS-Channel Binding - Holder of Key Das *Holder of Key* Verfahren wird in [22] vorgestellt und wird verwendet, um eine Sigle Sign-On Assertion an ein Client Zertifikat zu binden. Für dieses Verfahren wird ein einzelnes Client Zertifikat benötigt, dass für alle Verbindungen verwendet wird.
Ziel: Binden eines SAML-Tokens an eine TLS-Session in Verbindung mit einem Client Zertificates.
Voraussetzungen: Es wird davon ausgegangen, dass alle Parteien bereits über alle notwendigen Zertifikate verfügen. Ebenso wird davon ausgegangen, dass der Benutzer bereits über ein Konto beim Identity Provider verfügt und dieses mit einem Konto beim Service Provider verknüpft hat.
Verwendete Funktionalitäten: ***[KEYCALL]***, ***[SIGN]***, ***[DIGEST]***. Im User Agent werden keine kryptografischen Operationen benötigt, außer zur Erstellung des Zertifikates.
Vorgehen:

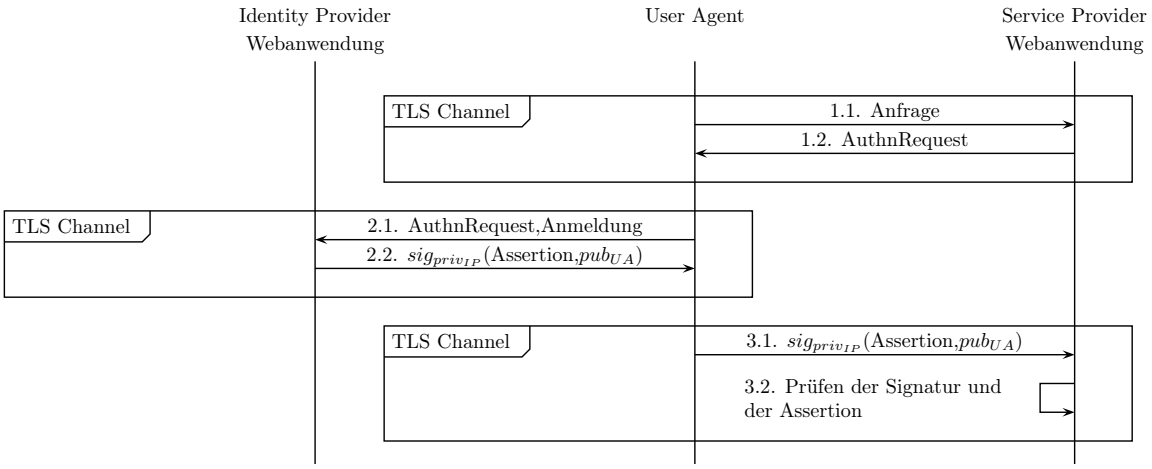


Abbildung 5.3.: Holder of Key, [22]

1. Der Anwender stellt eine Anfrage an die Webanwendung, diese erstellt eine Assertion für die Authentifizierung des Anwenders beim Identity Provider.

2. Der User Agent leitet die Anfrage weiter an den Identity Provider, dieser erstellt nach Authentifizierung des Benutzers einen Assertion und signiert diesen zusammen mit dem Schlüssel pub_{UA} aus dem verwendeten Client Zertifikat, zurück an den User Agent. ***[KEYCALL]***, ***[SIGN]***, ***[DIGEST]***

3. Der User Agent leitet den Assertion samt Signatur weiter zum Service Provider. Dieser prüft die Signatur und den Assertion. Durch die Einbettung des öffentlichen Schlüssels pub_{UA} kann die Webanwendung nun verifizieren, ob es sich um dieselbe Verbindung handelt, für die das Token ausgestellt wurde.

Dieses Verfahren ist auf die Verwendung eines Zertifikates für alle Verbindungen ausgelegt. Wenn nun

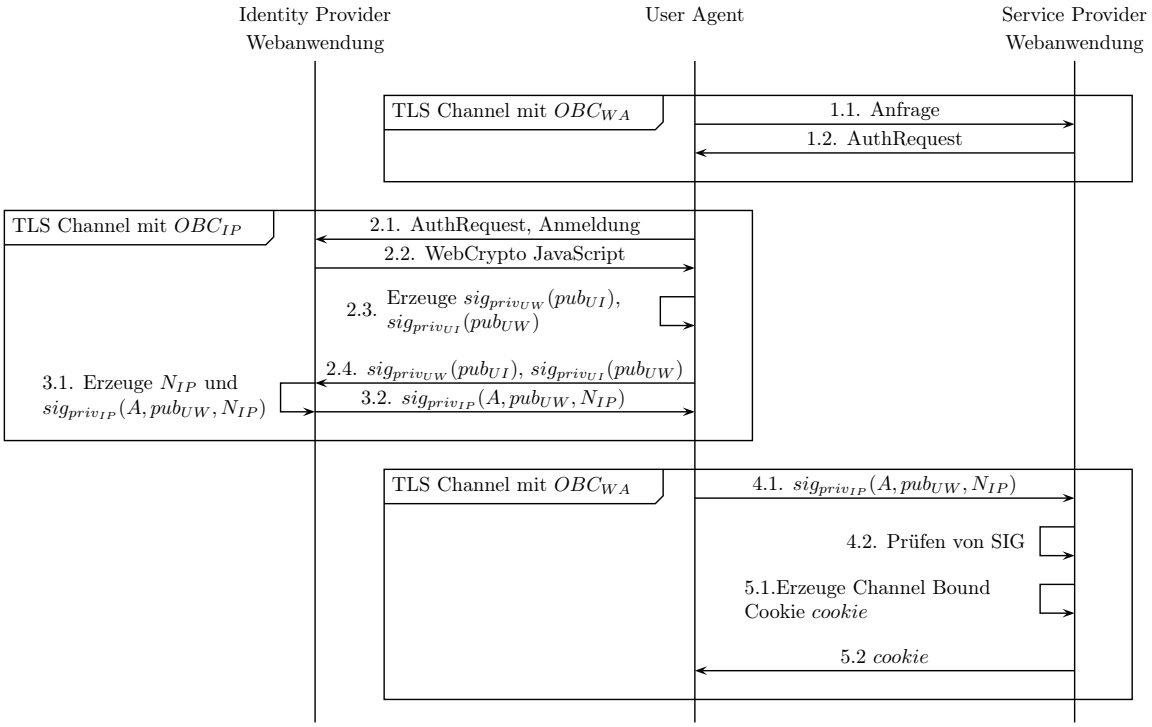
aber für jede Verbindung ein eigenes Zertifikat existiert, ist dieses Verfahren nicht mehr anwendbar, da der Identity Provider in Schritt 2.2. den öffentlichen Schlüssel aus der aktuellen Verbindung verwenden würde. Dieser Schlüssel ist ein anderer Schlüssel als der der für die Verbindung mit dem Service Provider verwendet wird. Aus diesem Grund würde der Service Provider die Assertion in ablehnen.
Das gleiche Prinzip, das Binden von Assertions an Client Zertifikate, wird in [21] für die Verwendung mit Origin-Bound Certificates eingeführt und in Abschnitt 5.2.1.2.2 beschrieben.

5.2.1.2.2. TLS-Channel Binding -Origin Bound Certifikates In [21] wurde ein Mechanismus vorgestellt, der Origin-Bound Certificates für Single Sign-On nutzt. So soll sicher gestellt werden, dass die Anfrage an den Service Provider und die Identity Provider von demselben Browser stammen. Diese Funktionalität soll nun mit Hilfe der WebCrypto API erreicht werden. Zu beachten ist, dass das spezifizierte Protokoll davon ausgeht, dass der User Agent selbständig die nötigen Operationen durchführt. Da allerdings bei der WebCrypto API kryptografische Operationen von der Webanwendung angestoßen werden, wurden im Diagramm Schritte hinzugefügt, die die nötigen JavaScript Methoden an den User Agent übermitteln.

Ziel: Es soll vom Identity Provider ein Token ausgestellt werden, dass es dem Client ermöglicht, sich beim Service Provider anzumelden. Dieses Token soll an eine bestimmte TLS-Verbindung gebunden werden. Mit Hilfe von *Cross Certification*, welche in [24] beschrieben wird. Bei der *Cross Certification* signiert der User Agent seinen öffentlichen Schlüssel aus dem Origin Bound Certifikates, der Verbindungen zum Identity Provider mit seinem privaten Schlüssel aus dem Origin Bound Certifikate, der Verbindung mit dem Service Provider und umgekehrt. So können sich sowohl der Service Provider als auch der Identity Provider sicher sein, dass die Anfrage von demselben User Agent kommt.
Voraussetzungen: Es wird davon ausgegangen, dass alle Parteien bereits über alle notwendigen Zertifikate verfügen. Ebenso wird davon ausgegangen, dass der Benutzer bereits über ein Konto beim Identity Provider verfügt und dieses mit einem Konto beim Service Provider verknüpft hat. Des Weiteren wird vorausgesetzt, das der Identity Provider den Namen kennt, unter dem der Service Provider den privaten Schlüssel des User Agents im User Agent abgelegt hat.
Verwendete Funktionalitäten: ***[KEYCALL]***, ***[VERIFY]***, ***[SIGN]***, ***[DIGEST]***, ***[NAMEDKEY]***, ***[RANDOM]***
Vorgehen: Der Ablauf einer solchen Anmeldung ist in Abb. 5.4 zu sehen. Die Bezeichnung *TLS Channel mit* OBC_i soll darauf hinweisen, dass der User Agent eine TLS Verbindung zu der Partei i, mit dem entsprechenden Origin-Bound Certificate aufgebaut hat.

1. Der Nutzer ruft die Webanwendung auf, die mit einem $AuthRequest$ antwortet.

2. Der User Agent leitet den $AuthRequest$ an den Identity Provider weiter und der Benutzer authentifiziert sich gegenüber dem Identity Provider. Dieser übermittelt das WebCrypto JavaScript an den User Agent. Der User Agent erzeugt nun die beiden Signaturen $sig_{privUW}(pub_{UI})$ und $sig_{privUI}(pub_{UW})$. Damit bindet der User Agent seinen öffentlichen Schlüssel aus der Verbindung zum Service Provider an seine Verbindung zum Identity Provider und umgekehrt. Zu beachten ist, dass der Identity Provider



Abweichende Bezeichnungen:

OBC_{WA}	:	Origin Bound Certifikate für die Verbindung zwischen User Agent und Webanwendung
pub_{UW} / $priv_{UW}$	:	Öffentlicher bzw. privater Schlüssel aus OBC_{WA}
OBC_{IP}	:	Origin Bound Certifikate für die Verbindung zwischen User Agent und Identity Provider
pub_{UI} / $priv_{UI}$	:	Öffentlicher bzw. privater Schlüssel aus OBC_{IP}
A	:	Single Sign-On Assertion

Abbildung 5.4.: Single Sign-On mit Origin-Bound Certificates, [21]

wissen muss, unter welchem Namen die Webanwendung des Serviceproviders den privaten Schlüssel aus dem OBC_{WA} im User Agent abgelegt hat. Dies ist nötig, da der Identity Provider den Namen des Schlüssel an den User Agent übergeben muss, damit er die Signatur $sig_{privUW}(pub_{UI})$ erzeugen kann. Im Folgenden wird die Nachricht $sig_{privUW}(pub_{UI}), sig_{privUI}(pub_{UW}), pub_{UI}, pub_{UW}$ an den Identity Provider übermittelt. ***[SIGN]***, ***[DIGEST]***, ***[NAMEDKEY]***

3. Der Identity Provider prüft nun die Signatur $sig_{privUI}(pub_{UW})$. Wenn diese Signatur gültig ist, kann der Identity Provider sicher sein, dass der User Agent eine Assertion zu der Verbindung anfordert, die zu pub_{UW} gehört. Weiter erzeugt der Identity Provider eine Nonce N_{IP}. Diese Nonce soll dem Service Provider ermöglichen die Freshness des Protokolllaufes zu prüfen. Anschließend erzeugt der Identity Provider die Signatur $sig_{priv_{IP}}(A, pub_{UW}, N_{IP})$, wobei A eine Assertion im Single Sign-On Umfeld darstellt. Nun wird die Nachricht $sig_{priv_{IP}}(A, pub_{UW}, N_{IP}), A, pub_{UW}, N_{IP}$ an den User

Agent weiter geleitet. ***[DIGEST]***, ***[VERIFY]***, ***[SIGN]***, ***[RANDOM]***, ***[KEYCALL]***

4. Der User Agent leitet die Nachricht $sig_{priv_{IP}}(A, pub_{UW}, N_{IP}), A, pub_{UW}, N_{IP}$ an die Webanwendung weiter. Diese überprüft nun die Signatur $sig_{priv_{IP}}(A, pub_{UW}, N_{IP})$. Anhand des Nachrichtenteils pub_{UW}, der vom Identity Provider mit signiert wurde, kann der Service Provider feststellen, ob die Anfrage und die Assertion vom selben User Agent stammen. Dies ist möglich, da der User Agent den öffentlichen Schlüssel der Verbindung mit dem Service Provider mit seinem privaten Schlüssel signiert hat, der zu der Verbindung mit dem Identity Provider gehört. ***[DIGEST]***, ***[VERIFY]***

5. Anschließend wird ein, wie in Abschnitt 5.2.1.1 beschrieben, ein Channel-Bound Cookie $cookie$ erzeugt, an den User Agent übermittelt und dort gespeichert.

5.3. Soft Tokens

Soft Tokens generieren ein Einmal-Passwort, das zusätzlich zu Benutzernamen und Passwort bei Anmeldevorgängen angegeben werden muss. Angenommen ein Benutzerkonto wird nur durch einen Benutzernamen und ein Passwort geschützt. Wenn nun ein Angreifer es schafft diese Werte auszuspähen, kann er sich an das Benutzerkonto anmelden. Wenn zusätzlich Einmal-Passworte verwendet werden, kann er das nicht mehr. Selbst wenn es ihm gelingt bei einer Anmeldung Benutzername, Passwort und Einmal-Passwort auszuspähen kann er sich nicht Anmelden, da er aus einem bekannten Einmal-Passwort keine weiteren berechnen kann. Einmal-Passworte werden, in der Regel, auf externen Geräten wie Mobiltelefonen oder spezial Hardware erstellt.
Ein verbreiteter Standard für die Erzeugung solcher Codes ist der *RFC 2289* [25], welcher die Erzeugung einer Hash-Kette vorsieht. Der Ablauf der Erzeugung und Verwendung von Einmal-Passworten (OTP) wird in Abb. 5.5 beschrieben. Die Sicherheit des Verfahrens basiert darauf, dass es sehr schwer ist eine Hash Funktion umzukehren.

Ziel: Erzeugung eines Einmalpasswortes
Voraussetzungen: Der Benutzer verfügt über ein Konto, bei der Webanwendung und möchte nun zusätzlich OTP verwenden. Es besteht die Möglichkeit eine Liste mit Hash-Werten zu erstellen und auf diese zuzugreifen.
Verwendete Funktionalitäten: ***[RANDOM]***, ***[DIGEST]***
Vorgehen:

1. Bei der Initialisierung des Verfahrens ruft der Nutzer die Webanwendung auf und meldet sich mit seinem Benutzernamen und Passwort an. Die Webanwendung übermittelt nun die nötigen JavaScript Methoden an den User Agent.

2. Der User Agent erzeugt nun einen Zufallswert K und berechnet $S = HASH(K)$, der Wert S wird in einer Liste abgelegt. In nächsten Schritt wird $S = HASH(S)$ erstellt und S wird an die Liste ange-

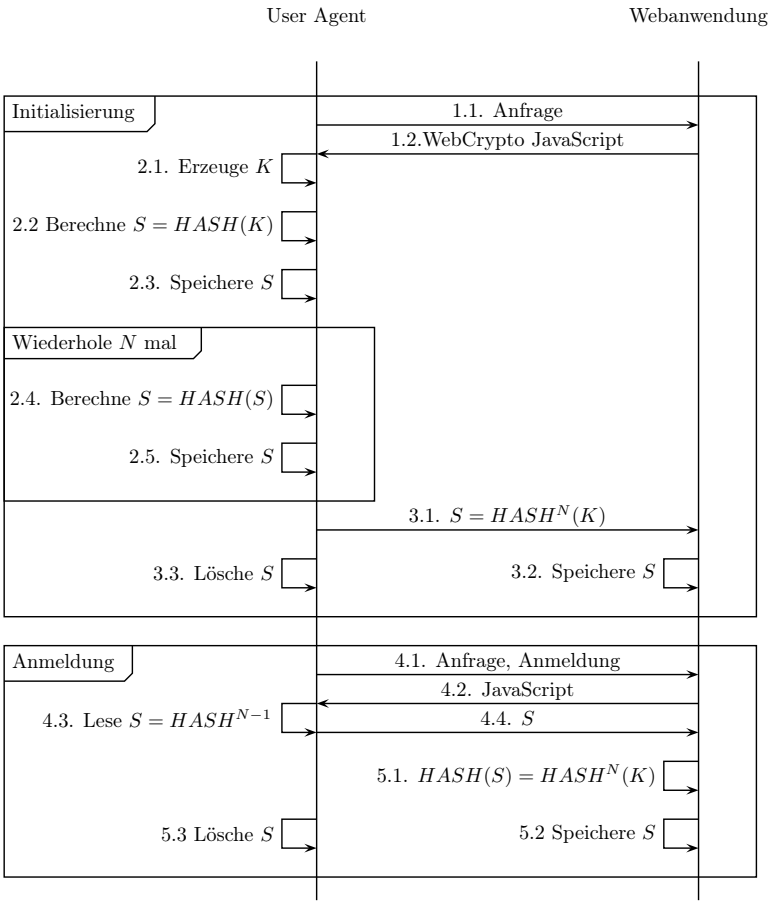


Abbildung 5.5.: One Time Passwort, [25]

hängt. Dieser Vorgang wird N-mal wiederholt. Als Ergebnis erhält man eine Liste mit N Hashwerten. ***[RANDOM]***, ***[DIGEST]***

3. Der Wert $S = HASH^N(K)$, also das N-malige Anwenden der Hash-Funktion auf den Wert K, wird an die Webanwendung übermittelt und dort gespeichert. Parallel wird dieser Wert aus der Liste des User Agents gelöscht da dieser Wert dort nicht mehr benötigt wird.

Im Folgenden wird nun ein OTP bei der Anmeldung an den Dienst verwendet.

4. Der Benutzer ruft die Webanwendung auf und meldet sich an, darauf hin sendet die Webanwendung den benötigten JavaScript Code an den User Agent. Der User Agent liest den letzten Hash-Wert aus seiner Liste, dies ist der Wert $S = HASH^{N-1}(K)$, und sendet diesen an die Webanwendung.

5. Die Webanwendung führt einmal die Hash-Funkion auf den Wert $S = HASH^{N-1}(K)$ aus und erhält so den Wert $HASH^{N-1}(K)$ und vergleicht diesen mit dem in Schritt 3.1. übermittelten Wert. Die

Webanwendung überschreibt nun den gespeicherten Wert mit dem eben übertragenen Wert. Der User Agent löscht $S = HASH^{N-1}(K)$ aus seiner Liste, der letzte Wert in der Liste ist nun $HASH^{N-2}(K)$. ***[DIGEST]***

Wenn ein Angreifer den in Schritt 3.1. oder 4.4. übertragenen Hash-Wert abfängt, kann er daraus keinen Nutzen ziehen, da er nicht den vorherigen Hash-Wert berechnen kann, der für die nächste Anmeldung benötigt würde. Wenn alle N erzeugten Werte aufgebraucht sind, wird das Verfahren neu initialisiert.

6. Implementierungen

6.1. Implementierung im Browser

In diesem Kapitel wird der Fortschritt der Implementierung der API im Browser untersucht. Das Augenmerk liegt dabei auf den Desktopversionen der Browser *Microsoft Internet Explorer*, *Mozilla Firefox* und *Google Chrome*, welche die größte Verbreitung haben. Über die Implementierung der API im *Apple Safari* Browser liegen keine Informationen vor, da *Apple* der einzige hier genannte Browser Anbieter ist, der sich nicht am Spezifizierungprozess beteiligt.

6.1.1. Microsoft Internet Explorer

Der Internet Explorer (IE) ist der einzige Browser, der bereits eine in [26] dokumentierte Implementierung der API enthält. Diese wurde in der *IE 11 Developer Preview* vorgenommen, basieren jedoch auf einem nicht näher genannten *älterem Editors Draft*. Da die Dokumentation allerdings das Datum von 13.04.2013 trägt, darf davon ausgegangen werden, dass es sich um einen sehr frühen Entwurf handelt, der vor dem ersten *public Working Draft* entstanden ist. Die bereits implementierten Algorithmen sind zusammen mit den unterstützen Methoden in Tab. 6.1 zusammengefasst.

Algorithmus	**enc**	**dec**	**sig**	**ver**	**dig**	**gen**	**imp**	**exp**	**wrp**	**uwrp**
RSAES-PKCS1-v1_5	✓	✓				✓	✓	✓		
RSASSA-PKCS1-v1_5			✓	✓		✓	✓	✓		
RSA-OAEP	✓	✓				✓	✓	✓		
AES-CBC	✓	✓				✓	✓	✓		
AES-GCM	✓	✓				✓	✓	✓		
AESKW									✓	✓
HMAC			✓	✓		✓	✓	✓		
SHA-1					✓					
SHA-256 / 384 / 512					✓					

Abkürzungen:

enc : `encrypt()` **ver** : `verify()` **im** : `importKey()` **uwrp**: `unwrapKey()`
dec : `decrypt()` **dig** : `digest()` **exp** : `exportKey()`
sig : `sign()` **gen** : `generateKey()` **wrp** : `wrapKey()`

Tabelle 6.1.: Algorithmen Internet Explorer, [26]

Wie aus dieser Aufstellung zu erkennen ist, unterstützt der Internet Explorer nicht alle zur Zeit empfohlenen Algorithmen (vgl. Tab. 3.10). Ebenso werden die Operationen `deriveKey()` und `deriveBit()` nicht unterstützt. Da keine Informationen vorliegen, auf welchem *Editors Draft* diese Implementierung basiert, lässt sich zu diesem Zeitpunkt nicht beurteilen, in wie weit man diese als Referenz für Prototypenentwicklung von Webanwendungen verwenden kann. Sie gibt allerdings Entwicklern die Möglichkeit, sich bereits jetzt mit der API vertraut zu machen.

6.1.2. Mozilla Firefox und Google Chrome

Sowohl bei Mozilla als auch bei Google existiert keine Dokumentation zu dem Fortschritt.
Mozilla Firefox: Hier existiert lediglich ein Bugtracker in dem keine Informationen zu konkreten Implementierungen zu finden sind. In wie fern bereits Methoden implementiert wurden, oder was konkret für die Zukunft geplant ist, ist nicht ersichtlich. [27]
Google Chrome: Bei Google existiert bisher ein Issue zu dem Thema, aus dem ersichtlich ist, dass bereits Arbeiten stattgefunden haben. Eine Übersicht existiert hier ebenfalls nicht, den Kommentaren der Entwickler ist allerdings zu entnehmen, dass bereits diverse Methoden und Algorithmen implementiert worden sind. Um eine Aufstellung zu liefern, müsste an dieser Stelle der Quellcode analysiert werden, was im Rahmen dieser Arbeit nicht möglich ist. [28]

6.2. Prototypen Webanwendungen

Offiziell existieren noch keine Prototypen von Webanwendungen, welche die WebCrypto API verwenden. An dieser Stelle sei allerdings auf das *Polycrypt* Project [29] hingewiesen, welches den *Editors Draft vom 17.12.2012* komplett in JavaScript implementiert.

7. Fazit

Die Idee, die hinter der *Web Cryptography API* steht, ist ein guter Ansatz, um kryptografische Operationen des Browsers für Webentwickler zugänglich zu machen. Es existieren jedoch noch ein paar Probleme. Ein besonderes Problem ist das Problem des Schlüsselzuordnung. Ein Ansatz, in dem der User Agent den passenden Schlüssel auswählen kann, wäre in der Praxis besser zu verwenden, da auf diese Art und Weise bereits kryptografische Operationen durchgeführt werden könnten, bevor der Benutzer sich an der Webanwendung anmeldet. So könnte z.B. der User Agent eine Challange an die Webanwendung stellen, bevor der Benutzer die Möglichkeit hat Daten an eine potentiell bösartige Webanwendung weiter zu geben. Darüber hinaus wäre so auch eine Anmeldung ausschließlich mit Schlüsselmaterial denkbar, welches dann z.B. Cookies ersetzen könnte.
Ein weiterer kritischer Punkt ist aus die Auswahl an Algorithmen. Da es keine Vorgaben gibt, welche Algorithmen ein standardkonformer User Agent implementieren muss, besteht die Möglichkeit, dass eine Webanwendung einen Algorithmus fordert, den ein bestimmter User Agent nicht unterstützt. Dies könnte dazu führen, dass Webanwendungen nur ein paar wenige Algorithmen verwenden, die jeder User Agent mit großer Wahrscheinlichkeit unterstützt. Es steht zu befürchten, dass aus diesem Grund vorwiegend ältere, weit verbreitete Algorithmen verwendet werden. Ein weiterer Punkt in diesem Kontext ist auch, dass User Agent und Webanwendung keine Algorithmen aushandeln können. Die Webanwendung gibt den Algorithmus vor, wenn der User Agent diesen nicht unterstützt, können die entsprechenden Funktionalitäten nicht eingesetzt werden. Dies ist ein weiterer Punkt weshalb die Verwendung neuer Algorithmen gehemmt werden könnte.

Im Ganzen gesehen, können durch die API Funktionen vom Browser übernommen werden für die bisher spezielle Anwendungen benötigt werden. Beispiele dafür die verschlüsselte Ablage von Dokumenten in der Cloud und die Verschlüsselung und Signierung von E-Mail direkt im Browser. Bei anderen Anwendungsmöglichkeiten wie, der Authentifikation oder dem ersetzten von Benutzernamen/Passwort oder Cookies, muss aufgrund des Schlüsselzuordnungsproblems noch nachgebessert werden.

Es bleibt abzuwarten wie, sich der Standard in der Zukunft entwickeln wird, da es sich um einen Standard handelt, der noch nicht final ist, jedoch Potential besitzt.

A. Anhang

A.1. Abhängigkeiten

A.1.1. DOM Future

In der Version 4 des *Data Object Models* (DOM), welche in [30] spezifiziert wird, wird das *future-* oder auch *promise-Pattern* spezifiziert. Dieses Pattern ermöglicht es, asynchrone Operationen durchzuführen. Ein *future* hat einen *Status*, ein *Ergebnis* und einen zugeordneten *Resolver*. Der *Resolver* ist ein Interface, dessen Instanziierung die eigentliche Operation durchführt. Das *Resolver Interface* beschreibt die folgenden Methoden:

- **`fulfill`**: Beim Aufruf dieser Methode wird der Status des Resolvers auf *Resolved* gesetzt und das Ergebnis der Operation wird übergeben.
- **`resolve`**: Wird diese Methode aufgerufen, wird die eigentliche Operation durchgeführt, beim Ereignis `onSuccess` wird das Ergebnis an die Funktion `fulfill` übergeben.
- **`reject`**: Setzt den Status auf *rejected* und gibt als Ergebnis den Eingabewert zurück. Diese Methode wird beim auftreten des Ereignisses `onError` aufgerufen.

Glossar

Anmeldetoken Siehe Token. 44

API Kurz für *application programming interface*. Definiert Schnittstellen zu einer Anwendung, die von anderen Anwendungen verwendet werden können. 1

CSRF Kurz für auch XSRF *Cross-Site Request Forgery*. Ein Angriff, bei dem Anwender ungewollt Aktionen in einer Webanwendung durchführt bei der sie zu diesem Zeitpunkt, angemeldet sind. 6

Enumeration Konstrukt, das einem Objekt eine bestimmte Menge an Werten zuordnet, die es annehmen kann. 19

First in first out Das Erste eingehende Objekt wird als erstes Bearbeitet. Beispiel: Warteschlange. 18

GUID Kurz für *Global Unique Identifier*. Beschreibt eine Bytefolge, die basierend auf ihrem Wertebereich weltweit als Einzigartig angenommen wird. 14

Identity Provider Anbieter, der Nutzerauthentifizierung im Rahmen von Single-Sign On anbietet. 44

Interface Vorschrift für den Aufbau eines Objektes in der Objekt-Orientierten Programmierung. 11

JavaScript Engine Laufzeitumgebung für JavaScript im User Agent. 10

Key Reuse Das Verwenden eines Schlüssels für mehrere Zwecke. 8

LowerCase Repräsentation einer Zeichenfolge nur in Kleinbuchstaben. 26

Nonce Eine Zufallszahl, die einen Wertebereich abdeckt, der es unwahrscheinlich macht, dass eine Zahl mehrfach verwendet wird. 30

PlugIn Programm, welches den Funktionsumfang einer Anwendung erweitert. 5

Service Provider Anbieter eines Dienstes im Internet. 15

Sigle-Sign On Es wird nur noch ein Benutzername/Passwort für mehrere Webanwendungen verwendet. Die Anmeldung erfolgt bei einem Identity Provider, der die Authentifikation gegenüber angeschlossenen Service Providern übernimmt. 44

TLS TLS und SSL sind Protokolle für die gesicherte Kommunikation zwischen Client und Server. TLS ist der Nachfolger von SSL. 6

Token Objekt, das eine Anmeldung mit bestimmten Rechten darstellt. 53

Unicode Standard zur Darstellung von Zeichen. 15

XSS Kurz für *Cross-Site-Scripting*. Ausnutzen einer Sicherheitslücke, in dem Informationen aus einem Kontext, in dem sie nicht vertrauenswürdig sind, in einen anderen Kontext eingefügt werden, in dem sie als vertrauenswürdig eingestuft werden [31]. 6

Literaturverzeichnis

[1] C. Paar and J. Pelzl, *Understanding Cryptography*. Springer, 2010.

[2] R. Sleevi, *Web Cryptography API, Editors Draft*, W3C Std., Aug. 2013. [Online]. Available: https://dvcs.w3.org/hg/webcrypto-api/raw-file/tip/spec/Overview.html

[3] Diverse, "Offizielle Mailing-List." [Online]. Available: http://lists.w3.org/Archives/Public/public-webcrypto/

[4] Harry Halpin, "Re-igniting the Crypto Wars on the Web," Dec. 2013. [Online]. Available: http://media.ccc.de/browse/congress/2012/29c3-5374-en-re_igniting_the_crypto_wars_on_the_web_h264.html

[5] Bundesamt für Sicherheit in der Informationstechnik, "Gefahren und Risiken im Umgang mit JavaScript/JScript." [Online]. Available: https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/Themen/Sicherheitsvorfaelle/AktiveInhalte/definitionen/javascriptgefahren.html

[6] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, , and L. F. Cranor, "Crying Wolf: An Empirical Study of SSL Warning Effectiveness." [Online]. Available: https://www.usenix.org/legacy/event/sec09/tech/full_papers/sunshine.pdf

[7] C. Boyd and A. Mathuria, *Protocols for Authetication and Key Establishment*. Springer, 2003.

[8] M. Watson, *Web Cryptography API Key Discovery, Editors Draft*, W3C Std., July 2013. [Online]. Available: https://dvcs.w3.org/hg/webcrypto-keydiscovery/raw-file/tip/Overview.html

[9] w3schools, "The Window Object." [Online]. Available: http://www.w3schools.com/jsref/obj_window.asp

[10] The Web Hypertext Application Technology Working Group, "HTML," Sept. 2013. [Online]. Available: http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

[11] B. Kaliski, *Public-Key Cryptography Standards (PKCS) 8: Private-Key Information Syntax Specification Version 1.2*, IETF RFC 5208, May 2008. [Online]. Available: http://tools.ietf.org/html/rfc5208

[12] I. X. P. K. I. Certificate and C. R. L. C. Profile, *The TLS Protocol*, IETF RFC 5280, May 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[13] (2013, Sept.) Web cryptography api use cases. [Online]. Available: https://dvcs.w3.org/hg/webcrypto-usecases/raw-file/tip/Overview.html

[14] I. Goldberg and J. Appelbaum, *Off-the-Record Messaging Protocol version 2*, The OTR Development Team Std., Sept. 2012. [Online]. Available: http://www.cypherpunks.ca/otr/Protocol-v2-3.1.0.html

[15] Zentraler Kreditauschuss, "ZKA-Kompendium Online-Banking-Sicherheit." [Online]. Available: http://www.hbci-zka.de/dokumente/diverse/ZKA%20Kompendium%20Online-Banking-Sicherheit%20V1.1%20final%20version.pdf

[16] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On Breaking SAML: Be Whoever You Want to Be." [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final91.pdf

[17] *SAML*, Oasis Std. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security

[18] millsd, Mozilla, "Introducing BrowserID: A better way to sign in." [Online]. Available: http://identity.mozilla.com/post/7616727542/introducing-browserid-a-better-way-to-sign-in

[19] M. Jones, J. Bradley, and N. Sakimura, *The TLS Protocol*, IETF Std., July 2013. [Online]. Available: http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-11

[20] T. Dierks and C. Allen, *The TLS Protocol*, IETF RFC 2246, Jan. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2246.txt

[21] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach, "Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web," 2012. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final162.pdf

[22] N. Klingenstein and T. Scavo, *SAML V2.0 Holder-of-Key Web Browser SSO*, OASIS Std., Aug. 2010. [Online]. Available: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-holder-of-key-browser-sso.pdf

[23] I. Hickson, *HTML*, W3C Std., Sept. 2013. [Online]. Available: http://www.whatwg.org/specs/web-apps/current-work/multipage/the-button-element.html#the-keygen-element

[24] Jim Turnbull, "Cross-Certification and PKI Policy Networking," Aug. 2000. [Online]. Available: https://www.netrust.net/docs/whitepapers/cross_certification.pdf

[25] N. Haller, C. Metz, and P. Nesser, "A One-Time Password System," IETF, Feb. 1998. [Online]. Available: http://tools.ietf.org/html/rfc2289

[26] Microsoft, "Web Cryptography API," Feb. 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/ie/dn265046%28v=vs.85%29.aspx

[27] Diverse, "Bug 865789 - (web-crypto) Implement W3C Web Crypto API ." [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=865789

[28] Diverse, "Issue 245025: Implement WebCrypto." [Online]. Available: https://code.google.com/p/chromium/issues/detail?id=245025

[29] BBN Technologies, "PolyCrypt: A WebCrypto Polyfill." [Online]. Available: http://polycrypt.net/

[30] A. van Kesteren, A. Gregor, and Ms2ger, *DOM*, W3C Std., Sept. 2013. [Online]. Available: http://dom.spec.whatwg.org/

[31] Bundesamt für Sicherheit in der Informationstechnik, "Glossar Cross-Site Scripting." [Online]. Available: https://www.bsi-fuer-buerger.de/BSIFB/DE/Wissenswertes_Hilfreiches/Service/Glossar/Functions/glossar.html?lv2=1661642&lv3=2148660

[32] J. Schwenk, F. Kohlar, and M. Amon, "The Power of Recognition - Secure Single Sign-On using TLS Channel Bindings," Aug. 2011. [Online]. Available: http://www.nds.rub.de/media/nds/veroeffentlichungen/2011/08/30/dim05k-schwenk.pdf

[33] J. Altman, N. Williams, and L. Zhu, *Channel Bindings for TLS*, IETF RFC 5929, July 2010. [Online]. Available: http://tools.ietf.org/html/rfc5929

[34] Bundesamt für Sicherheit in der Informationstechnik, "JavaScript/JScript Sicherheitsmodell." [Online]. Available: https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/Themen/Sicherheitsvorfaelle/AktiveInhalte/definitionen/javascriptsichmodell.html;jsessionid=63F725B1D786C30413E45467B71777A5.2_cid368