Transferleistung Theorie/Praxis* 2

Matrikelnummer:	8323
Freigegebenes Thema:	Weiterleitung der Zwei-Faktor-authentifizierten Anmeldung eines Versicherten aus der TK-App an die Webanwendung Meine TK
Studiengang, Zenturie:	A17b. Angewandte Informatik / 2017.

* *Studierende, die unter den Anwendungsbereich der PVO bis 03.02.2015 fallen, fertigen Transferleistungen weiterhin in der Form von Praxisberichten an und der Begriff hält Einzug in das Abschlusszeugnis. Ab dem Jahrgang 2016 hat der Begriff vollumfängliche Gültigkeit. In der Kommunikation hält der Begriff Transferleistungen ab sofort Einzug.*

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

THEORIE PRAXIS

Weiterleitung der Zwei-Faktor authentifizierten Anmeldung eines Versicherten aus der TK-App an die Webanwendung Meine TK.

Transferleistung
im Studiengang Angewandte Informatik
angefertigt im Projekt-Team Online and Mobile Projects
der Techniker Krankenkasse

Martrikelnummer: 8323

Hamburg, den 2. Januar 2019

Sperrvermerk

Die Inhalte dieser Arbeit betreffen Informationen über betriebliche Belange der Techniker Krankenkasse. Dies gilt ebenfalls bezüglich der für die Beurteilung der Arbeit zur Verfügung gestellten Dokumente der Techniker Krankenkasse. Die Arbeit selbst und weitere Dokumente sind für den innerbetrieblichen Gebrauch bestimmt und dürfen somit der Öffentlichkeit – auch in Auszügen – nicht zugänglich gemacht werden.

Inhaltsverzeichnis

Glossar

Grant Type bestimmt, wie Clients Authorization Grants von Resource Ownern erhalten können. Die vier Grant Types Authorization Code, Implicit, Resource Owner Password Credentials sowie Client Credentials wurden in RFC 6749 definiert; auch eigene Grant Types zu definieren ist möglich (vgl. Hardt 2012, S. 23ff.).. 2–4

Internet Engineering Task Force (IETF) Organisation von Workinggroups, die sich mit je einem spezifischen Thema befassen. Teil der Internet Society (ISOC). 2, 7

Minimal Viable Product (MVP) Umsetzung eines Features oder Produktes mit zwar minimalem Funktionsumfang, aber dennoch konkretem Mehrwert für den Nutzer. 1

OAuth 2.0 Protokoll zur API-basierten Autorisierung. 1–3, 7, 9

Transport Layer Security (TLS) Protokoll zur hybriden Verschlüsselung von Datenübertragungen, Grundlage für HTTPS. 3

Kapitel 1

Einleitung

Seit ungefähr 2016 erprobt die TK Softwareentwicklung in agilen Teams. Das Team Online and Mobile Projects, in dem diese Arbeit entstand, ist eines der Pilotprojekte dieses Vorstoßes. Das Team ist unter anderem mit der Betreuung und Weiterentwicklung der TK-App betraut. Die TK-App versucht, Verwaltungsprozesse, Kundeninteraktion im Allgemeinen, nutzerfreundlich umzusetzen.
In Meine TK, der Web-Plattform für Versicherte der TK, sind einige der von uns für die App geplanten Prozesse bereits existent. Teil der agilen Planung ist das Minimal Viable Product (MVP). MVPs sind minimal nutzbare Produkte, das bedeutet auch, das kleine Teile der Nutzergruppe, gerade in den ersten Zyklen der Umsetzung, nur teilweise abgebildet werden können. Denkbar ist es, einen Prozess im ersten Schritt nur teilweise in der App zu verwirklichen und den Nutzer ab einem bestimmten Punkt an Meine TK weiterzuleiten und den Prozess dort fortzusetzen. Hierzu müssen Nutzer sich derzeit stets erneut einloggen. Ziel dieser Arbeit ist, den Prozess der Weiterleitung für Nutzer angenehmer zu gestalten, indem die erneute Eingabe seiner Daten für ihn entfällt.
Die Weiterleitung von Zugriffsrechten ist seit 2012 mit der Veröffentlichung von OAuth 2.0 (Hardt 2012) grundsätzlich keine schwere Aufgabe mehr. Da die TK jedoch hohe Ansprüche an ihrer Sicherheitsstandards stellt und die Sicherheit der Implementierung eines Protokolls nicht durch die Sicherheit des Protokolls sichergestellt ist, wie (Sun & Beznosov 2012, Hu, Yang, Li & Lau 2014, Yang, Li, Lau, Zhang & Hu 2016) zeigten, ist diese Arbeit entstanden.
Die Frage, ob sich OAuth, das als Authentisierungsverfahren gedacht ist, für die Authentifizierung eines Nutzers eignet, wird in dieser Arbeit elegant umschifft.

Kapitel 2

OAuth 2.0

OAuth 2.0 ist ein von der Internet Engineering Task Force (IETF) konzipiertes Protokoll zur Delegation von Zugriffsrechten für Anwendungssoftware. Zugriffsrechte werden stets durch den Besitzer der Ressource, den Resource Owner, erteilt. Sie werden durch zeitlich limitierte Access-Tokens abgebildet. Access-Tokens lassen sich gut über den Vergleich mit Konzerttickets oder Bargeld erläutern. Wer im Besitz des zeitlich beschränkten Tickets ist, der Zeichenfolge des Tokens, muss sich nicht weiter ausweisen. Er ist so auch im Besitz der mit dem Token einhergehenden Zugriffsrechte (vgl. Siriwardena 2014, S. 134). Beispielsweise dem Zugriffsrecht auf eine API, die Nutzerdaten zurückgibt. Access-Tokens sind also eine Möglichkeit für den Resource Owner, Drittanbietern limitierten Zugriff auf Daten zu ermöglichen, die für ihn bereits von einer Anwendung verwaltet werden. Das Zugriffsrecht geht dabei allein auf den Besitz des Tokens zurück. Auf Access-Tokens soll in Abschnitt 2.3 näher eingegangen werden. Die Übergabe des Access-Tokens ist durch sogenannte Grant Types definiert. Sie beschreiben die Interaktionen zwischen den OAuth-Parteien, die notwendig sind, um einen Access-Token zu erlangen. Die in dieser Arbeit genutzten Grant Types, Authorization Grant sowie Implicit Grant, werden in Abschnitt 2.2 erläutert. Die Kommunikation innerhalb der Grant Types erfolgt mittels standardisierter Nachrichten, die als JSON per HTTP-Methode ausgetauscht werden (vgl. Hardt 2012, S. 29ff.). Im Folgenden werden die in OAuth 2.0 beteiligten Parteien, Objekte und Grant Types erläutert.

2.1 OAuth-Parteien

Resource Owner	besitzt geschützte Ressourcen, die auf einem Resource Server gehostet werden.
Resource Server	hostet geschützte Ressourcen, nimmt Requests auf geschützte Ressourcen entgegen und gewährt Zugriff, sofern der übermittelte Access-Token valide ist.
Client Application	ist eine Anwendung, die Zugriff auf geschützte Ressourcen anfragt und entsprechende Requests an den Resource Server stellt. Jeder Client muss sich am Authorization Server registrieren, woraufhin ihm eine Client-ID und ein Client-Secret zugewiesen werden.
Authorization Server	authentifiziert den Resource Owner anhand seiner Zugangsdaten und stellt dem Client, je nach Grant Type, einen Authorization Code oder einen Access-Token auf die angefragte Ressource zur Verfügung. (vgl. Hardt 2012, S. 6)

2.2 Grant Types

Grant Types, auch Flows genannt, beschreiben die Interaktionen der vier Parteien untereinander, die notwendig sind, um Zugriff auf eine geschützte Ressource zu erhalten. Abbildung 2.1 stellt den grundlegenden Kommunikationsablauf zwischen dem Client auf der einen, sowie Resource Owner, Authorization Server und Resource Server auf der anderen Seite dar. Jegliche Netzwerkkommunikation zwischen den Parteien sollte hierbei unbedingt durch TLS geschützt werden (vgl. Siriwardena 2014, S. 99). Der Protocol Flow (Abbildung 2.1) lässt sich auf drei grundlegende Schritte reduzieren.

Authorization Grant — der Client stößt einen der vier spezifizierten Grant Types an. Interessant an Grants ist, dass Resource Owner-Credentials in den meisten Grant Types nur an den Authorization Server übermittelt werden, nicht an den Client.

Access-Token Retrieval — vorausgesetzt, der Grant war erfolgreich, erhält der Client einen Access-Token. In einigen Grant Types spiele Refresh-Tokens eine Rolle, in dieser Arbeit spielen sie keine…

Resource Access — durch Vorlage des Access-Tokens erhält der Client Zugriff auf die entsprechende Ressource.

Im Folgenden werden insbesondere der Authorization Code Grant sowie der Implicit Grant erläutert. Resource Owner Password Credentials Grant und Client Credentials Grant sind Reduktionen der beiden vorgestellten Grant Types und werden nicht weiter behandelt.

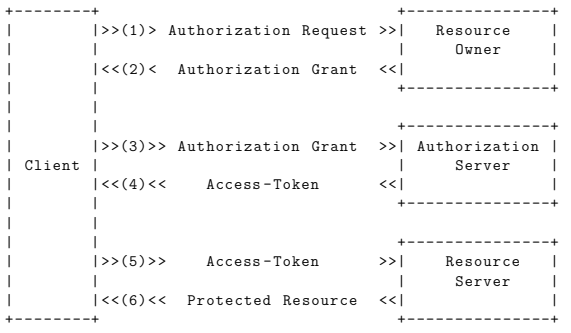

Abb. 2.1: Protocol Flow (vgl. Hardt 2012, S. 7)

2.2.1 Authorization Code Grant

Der Authorization Code Grant ist immer dann möglich, wenn der Client auf einen Web-Browser zugreifen kann und über ein eigenes Backend verfügt oder anderweitig dazu in der Lage ist, sein Client Secret geheimzuhalten (vgl. Hardt 2012, S. 24ff.). Zwar ist der Authorization Code Grant, aufgrund der vielen Authentifizierungszwischenschritte der umfangreichste, hierdurch aber auch der sicherste der in OAuth 2.0 genutzten Grant Types. Vorausgesetzt, die Übermittlung der Daten erfolgt per Transport Layer Security (TLS), ist die Zuverlässigkeit des Authorization Code Grant formal gesichert (Chari,

Jutla & Roy 2011). Wie schon der Protocol Flow (vgl. Hardt 2012, S. 7), lässt sich der Authorization Code Grant leicht herunterbrechen. Er verfolgt zwei zentrale Ziele: zum einen, den Erwerb eines Authorization Codes, zum anderen das Einlösen dieses Authorization Codes gegen einen Access-Token. Abbildung 2.2 zeigt den Authorization Code Flow.

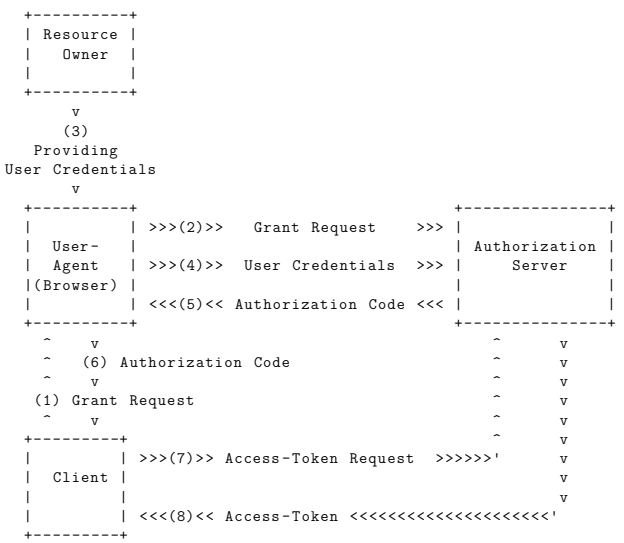

Abb. 2.2: Authorization Code Grant (vgl. Hardt 2012, S. 24)

(1–6) Authorization Code Retrieval

Um Zugriff auf einen Authorization Code zu erlangen, redirectet der Client den Resource Owner im ersten Schritt an den Authorization Server. Aufgabe des Authorization Servers ist es, die Identität des Clients sicherzustellen. In der Regel geschieht dies über einen Login am Authorization Server oder eine bereits bestehende Session, die Authentifizierung ist grundsätzlich aber nicht Thema des Protokolls. Wie in den meisten Grant Types, hat der Client selbst nie Zugriff auf die Credentials des Resource Owners. Der Redirect besteht aus zwei erforderlichen Query Parameter (vgl. Hardt 2012, S. 25ff.):

response_type um einen Authorization Code zu erhalten, muss der Wert ‘code’ gewählt werden.

client_id die Client ID wie in Kapitel 2.1 beschrieben. Der erhaltene Authorization Code kann nur mit dem zugehörigen Client Secret eingelöst werden.

Weitere Parameter wie redirect_uri, scope und state sind optional. Abbildung 2.3 zeigt einen beispielhaften Request. Vorausgesetzt, der Grant war erfolgreich, wird der User Agent an die bei der

```
https://localhost/authorize?
                    response_type=code&
                    client_id=my4Client2giz4IDb5zn
```

Abb. 2.3: Authorization Request (vgl. Siriwardena 2014, S. 97)

Client-Registrierung angegebene Redirect URI weitergeleitet, wobei der Authorization Code im URI Fragment oder als Query Parameter gesetzt ist (vgl. Hardt 2012, S. 25) (Abbildung 2.4). Wie der Client

auf den Parameter zugreift ist nicht Teil der Spezifikation. Zu beachten ist, dass der Authorization Code auf diesem Weg auch für den Nutzer und im Falle eines manipulierten Browsers auch für etwaige Angreifer sichtbar ist. Um Replay-Angriffen vorzubeugen, sollte jeder Authorization Code nur ein einziges Mal genutzt werden können (vgl. Siriwardena 2014, S. 97). Falls der Authorization Server eine erneute Nutzung desselben Authorization Codes feststellt, sollten alle bereits ausgestellten Access-Tokens invalidiert werden. (vgl. Siriwardena 2014, S. 97)

```
https://myuri/#code=45273487ckdjfhjgasduz123ioujuj
```

Abb. 2.4: Authorization Response (vgl. Siriwardena 2014, S. 98)

(7–8) Access-Token Retrieval

Abschließend muss der Client den erhaltenen Authorization Code gegen einen Access-Token eintauschen. Hierzu ist ein POST (Abbildung 2.5) auf den Access-Token-Endpoint des Authorization Servers auszuführen, der zwei erforderliche Parameter entgegennimmt (vgl. Hardt 2012, S. 29):

`grant_type` um zwischen dem Authorization Code Grant und weiteren Grant Types wie dem Resource Owner Password Credentials Grant zu unterscheiden. Als Wert muss ‘authorization_code’ gewählt werden.

`code` der im vorangegangenen Schritt erhaltene Authorization Code.

Der Access-Token Endpoint sollte mindestens per HTTP Basic Authentication abgesichert werden. Client ID und Client Secret dienen als Zugangsdaten. Da die Zugangsdaten in der Basic Authentication nicht verschlüsselt übertragen werden, ist eine Übermittlung ausschließlich per HTTPS anzuraten (vgl. Siriwardena 2014, S. 97) Nur falls der Client nicht in der Lage sein sollte, HTTP Basic Authentication auszuführen, dürfen Client ID und Client Secret im Body des Requests übertragen werden (vgl. Hardt 2012, S. 15f.).

```
curl -v -X POST --basic
     -u  my4Client2giz4IDb5zn:my32Client9btzSecret186baszdg1
     -H "Content-Type:application/x-www-form-urlencoded;charset=utf-8" -k
     -d "grant_type=authorization_code&
         code=45273487ckdjfhjgasduz123ioujuj" https://localhost/token
```

Abb. 2.5: Access-Token cURL (vgl. Siriwardena 2014, S. 98)

Auf das vorangegangene cURL-Kommando in Abbildung 2.5 sollte der Authorization Server mit einem HTTP 200 OK und einem JSON-Body (Abbildung 2.4) antworten (vgl. Hardt 2012, S. 31). Die möglichen Token Types werden in Kapitel 2.3 näher erläutert. Neben dem Access-Token enthält das JSON Informationen über die Gültigkeitsdauer des Access-Tokens in Sekunden sowie den weiter oben bereits kurz erwähnten Refresh-Token (ebd.), der in dieser Arbeit jedoch keine weitere Rolle spielt.

```
{
    "token_type":"bearer",
    "expires_in":"600",
    "refresh_token":"31zse4ua14fkjsd5bg39gsjnzg42zg",
    "access_token":"7c2hg718hn21sd2sz693jas6h24g5i"
}
```

Abb. 2.6: Access-Token Response (vgl. Siriwardena 2014, S. 98)

2.2.2 Implicit Grant

Anders als der Authorization Code Grant richtet sich der Implicit Grant (Abbildung 2.7 unsichere Clients, die über kein eigenes Backend verfügen und auch anderweitig nicht dazu in der Lage sind, ihr Client Secret zu schützen (vgl. Siriwardena 2014, S. 98). Da Authorization Codes nur mittels Client Secret in Access Tokens eingetauscht werden können, entfallen Authorization Codes für den Implicit Grant. Der Request an den Authorization Server (Abbildung 2.8 ist in seinem Aufbau ident zu

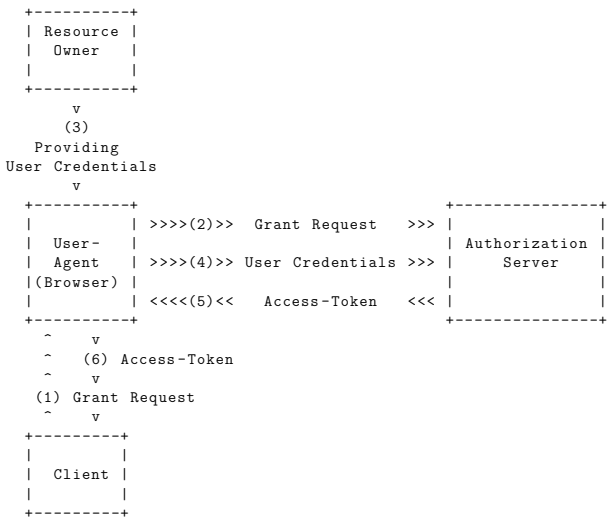

Abb. 2.7: Implicit Grant (vgl. Hardt 2012, S. 32)

Abbildung 2.3. Der Wert des response_type ist auf ‘token’ anzupassen (vgl. Hardt 2012, S. 19). Nach

```
https://localhost/authorize?
                  response_type=token&
                  client_id=my4Client2giz4IDb5zn
```

Abb. 2.8: Implicit Authorization Request (vgl. Siriwardena 2014, S. 98)

erfolgreicher Authentifizierung des Resource Owners in Schritt 1–4 antwortet der Authorization Server direkt mit einem Access-Token (Abbildung 2.9). Die Schritte 7–8 des Authorization Code Grants entfallen also. Refresh-Tokens, die in dieser Arbeit ohnehin keine weitere Rolle spielen, werden in Reaktion auf die geminderte Vertraulichkeit des Implicit Grants nicht mehr bereitgestellt.

```
https://myuri/#access_token=45273487ckdjfhjgasduz123ioujuj&
                expires_in=600
```

Abb. 2.9: Implicit Authorization Response (vgl. Siriwardena 2014, S. 99)

2.3 Access-Token

Bei Access-Tokens handelt es sich um serverseitig generierte, kryptographische Werte (vgl. Rohr 2018, S. 273), also zufällige Zeichenfolgen, die nur schwer erraten werden können (vgl. Petrlic & Sorge 2017, S. 72) Einer der zentralen Kritikpunkte an OAuth 2.0 ist die Ungebundenheit der in RFC 6750 spezifizierten Bearer-Tokens, wie sie bereits zu Anfang dieses Kapitels beschrieben wurden (vgl. Hammer 2012). Eine Alternative zu Bearer-Tokens stellen MAC-Tokens dar.

MAC-Token

MAC Tokens, spezifiziert in dem IETF HTTP MAC Draft, verhalten sich, um die bereits genutzte Geld-Metapher wieder aufzugreifen, wie Kreditkarten (vgl. Siriwardena 2014, S. 134). Hierzu wird dem Client ein symmetrischer Schlüssel übermittelt, mit dem er den MAC berechnet. Da MAC-Tokens in Kapitel 3 ausgeschlossen werden, sollen sie hier nicht weiter betrachtet werden. Zusätzliche Informationen bezüglich der Berechnung eines MAC-Tokens finden sich in (Siriwardena 2014, S. 136ff.) sowie in RFC 2104.

2.4 JSON Web Encryption

Definiert in RFC 7159 ist JavaScript Object Notation (JSON) ein Format zur Übertragung oder Speicherung von Daten, das sich in den vergangenen Jahren als Alternative zu XML etabliert hat (vgl. Siriwardena 2014, S. 201). JSON Web Token (JWT) ist ein JSON-basiertes Container-Format für Datenübermittlung, das insbesondere als Dateiformat in OpenID Connect genutzt wird (vgl. Sakimura, J., M., de Medeiros B. & C. 2014, Chapter 2, ID Token). JWTs werden in der Praxis entweder signiert (JWS) oder verschlüsselt (JWE). Im Folgenden soll ein beispielhafter JWE erstellt werden.
JWEs bestehen, je nach Anwendungsfall aus insgesamt fünf bis sechs Attributen (vgl. Siriwardena 2014, S. 213ff.). JWEs mit sechs Attributen können sich an mehr als nur einen Empfänger wenden (ebd.), was für diese Arbeit jedoch irrelevant ist. Der Inhalt (Payload) eines JWEs wird zunächst symmetrisch verschlüsselt (ebd.). Der hierzu genutzte Content Encryption Key wird anschließend asymmetrisch verschlüsselt und ist Teil des JWEs.

```
{
  "header":{"enc":"A128GCM","alg":"RSA-OAEP"},
  "encrypted_key":"G41Y2aizjBW5bFqEYWhIL84bVBbcuaHXB1szgFPEKfqJIMfcUZ
                   faeNqG2B9Cxo7Q",
  "initialization_vector":"3THSMF4n786zAUim204dhgknnJz85FkP",
  "ciphertext":"YXBKFvHo6nxQICZ3jTC1jIYhyHoDGwTDPuFzx2YgWCF39dP2Y6SFzCW24BzpiD0A",
  "tag":"gVYxhN5OVTAWcYub"
}
```

Abb. 2.10: JWE

Abbildung 2.10 zeigt ein fünfteiliges JWE mit den Attributen

header — enthält unverschlüsselte Metainformationen über die Verschlüsselung des Content Encryption Keys. Wichtige Elemente sind

- alg der Algorithmus zur Verschlüsselung des Content Encryption Keys.
- enc der Algorithmus zur Verschlüsselung des Payloads, also der zu übermittelnden Daten.

encrypted_key — base64url-encodierter Content Encryption Key, verschlüsselt durch das in ‘alg’ spezifizierten Algorithmus.

initialization_vector — base64url-encodierter, zufällig generierter Wert, der im Rahmen mancher Ver- und Entschlüsselungsverfahren benötigt wird

ciphertext — base64url-encodiert, enthält die zu übermittelnden Daten der Nachricht. Verschlüsselt durch den Content Encryption Key nach dem in ‘enc’ spezifizierten Algorithmus.

tag — Signatur, die bei der Verschlüsselung erstellt wird.

Zur Serialisierung werden die Attribute, falls sie es noch nicht sein sollten, base64url-Encodiert und mit Punkten getrennt konkateniert (Abbildung 2.11). Die Erstellung und Verschlüsselung eines JWEs

```
eyJlbmMiOiJBMTI4R0NNIiwiYWxnIjoiUlNBLU9BRVAifQ
.G41Y2aizjBW5bFqEYWhIL84bVBbcuaHXB1szgFPEKfqJIMfcUZfaeNqG2B9Cxo7Q
.3THSMF4n786zAUim204dhgknnJz85FkP
.YXBKFvHo6nxQICZ3jTC1jIYhyHoDGwTDPuFzx2YgWCF39dP2Y6SFzCW24BzpiD0A
.gVYxhN5OVTAWcYub
```

Abb. 2.11: Serialisiertes JWE

wird im Anhang in Abbildung A.3 demonstriert, die Entschlüsselung in Abbildung A.4. Abbildung A.1 stößt den Prozess als Main-Methode an.

Kapitel 3

Implementierung

Der in unserem Team aufgetretene Anwendungsfall involviert zwei Clients, die TK-App und den Web-Client, mit jeweils unabhängigen Backendsessions. Die TK-App ist durch Einbindung KOBILs als sicherer Client zu betrachten. Diesen Umstand zu elaborieren überschreitet den Umfang dieser Arbeit, er ist als gegeben hinzunehmen. Der Web-Client kommuniziert per HTTPS und nutzt secured-Cookies, durch seine Backendanbindung ist auch er als grundsätzlich sicher einzustufen. Bei erstmaligem Aufruf des Web-Clients wird im User-Agent des Nutzers ein neuer, noch nicht eingeloggter TKSESSION-Cookie gesetzt. Es wird angenommen, dass der Resource Owner bereits in der App eingeloggt ist. Zielsetzung ist nun, die im User-Agent des Web-Clients gesetzte Session einzuloggen. Zur Verfügung stehen die bereits aufgezählten Parteien: eingeloggte TK-App + Backend, User-Agent, Web-Client + Backend.
In ersten Überlegungen wurde die Erstellung einer neuen, eingeloggten Session in Betracht gezogen, was sich jedoch schnell als Vergehen am Loadbalancer herausstellte und daher verworfen wurde. Auch der Ansatz, die Session ID an die TK-App weiterzuleiten und in ihrem Backend einzuloggen erwies sich als undurchführbar. Um eine Session einzuloggen, muss ein Session-eigener User-Context befüllt werden, was durch eine außerhalb liegende Session unmöglich ist.
Die App selbst ist also nicht dazu in der Lage, die Web-Session einzuloggen. Der Login muss also über den Web-Client erfolgen. Durch diese Erkenntnis kann auf eine klassische OAuth 2.0-Problemstellung reduziert werden: der Web-Client greift auf eine Login-API zu. Um diese vor unbefugten Nutzern zu schützen, setzt der Zugriff auf sie einen Access-Token voraus. Da sich in der TK-Implementierung von dem Access-Token intern auf den Versicherten schließen lässt, für den er ausgestellt wurde, ist ein weiterer Parameter mit einer Nutzerkennung nicht erforderlich.

3.1 Entkoppelte Flows

Abbildung 3.1 zeigt eine mögliche Realisierung als von der App ausgehender Implicit Flow. Sie ist die einfachste Realisierung des Anwendungsfalls. Dieses Vorgehen weist jedoch zwei wichtige Fehler auf: zunächst dies: die einzige Verbindung zwischen App- und Web-Client, ist der in Schritt 3 übertragene Access-Token. Web-APIs können jedoch von jedem User-Agent aufgerufen werden. Es ist demnach anzunehmen, dass entsprechende APIs von Dritten angegriffen werden — und diese Angriffe erfolgreich sein werden, falls es diesen Dritten gelingen sollte, in den Besitz eines Access-Tokens zu gelangen. Hier also, ebenfalls in Schritt 3, findet sich der zweite Fehler. Die Übertragung des Access-Tokens an den Web-Client erfolgt unverschlüsselt als Query Parameter. Durch einen korrumpierten User Agent, respektive Smartphone, könnte es einem Dritten nun möglich sein, den Access-Token zu entwenden.

Da Tokens mehrfach nutzbar sind, können Dritte sich nun durch einen Aufruf der login/redirect-API mit dem entwendeten Token Zugang zu einer eingeloggten Session verschaffen. Eine erste Verbesserung

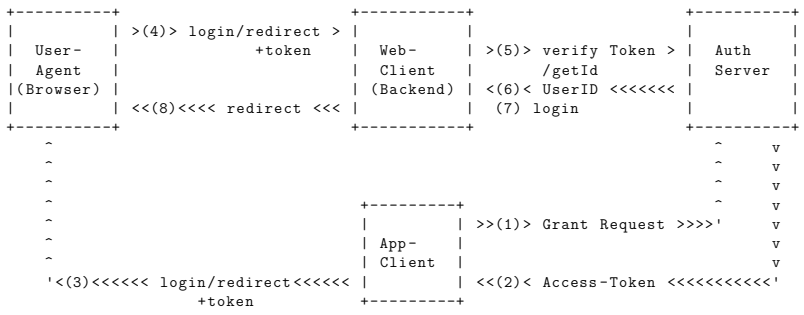

Abb. 3.1: Anwendungsfall, Implicit Login Flow

wäre es, in Schritt 3 statt des Access-Tokens einen Authorization Code zu versenden (Abbildung 3.1). Im Backend des Web-Clients ist die Umwandlung des Authorization Codes in einen Access-Token von nur geringem zusätzlichem Aufwand. Die ausgestellten Authorization Codes sollten in jedem Fall nur einmalige Gültigkeit besitzen, bei erneuter Nutzung sollten alle eingeloggten Sessions des Resource Owners invalidiert werden, um Replayangriffe blockieren zu können. Die Entwendung

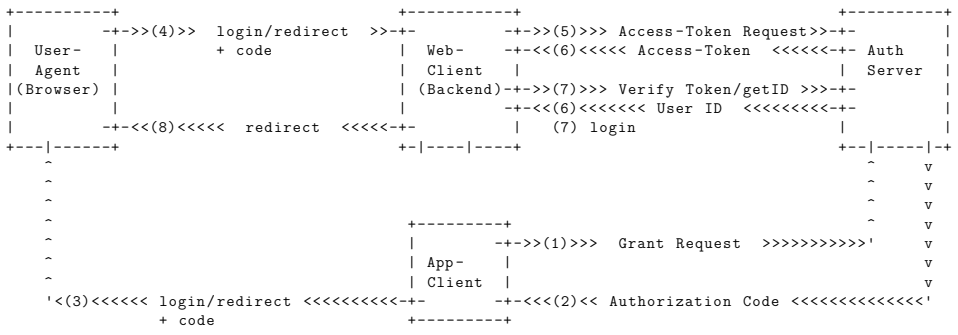

Abb. 3.2: Anwendungsfall, Authorization Code

des Parameters in Schritt 3 wird auf diese Weise zwar nicht verhindert, dadurch dass es sich jedoch nur um einen einmalig gültigen Authorization Code handelt, ist der Wert eines Diebstahls erheblich vermindert. Da betroffene Versicherte sich jedoch neu einloggen müssten, würde ein derartiger Angriff dem Nutzererlebnis dennoch schaden.

Um den Login eines Resource Owners in diesem Flow zu stehlen, wäre es notwendig, die Verbindung des User-Agents zu der Web-API zu unterbinden. In einem korrumpierten User-Agent, respektive Smartphone, wäre dies zwar wieder denkbar, dass eine Verbindung vollkommen blockiert wird, wirkt jedoch noch unwahrscheinlicher als der bloße Diebstahl übermittelter Daten. Davon abgesehen: einen geschickten Angriff dieser Art würde der Resource Owner nicht wahrnehmen, der korrumpierte User-Agent könnte ihn beispielsweise auf die Startseite der TK redirecten. Dass Resource Owner-Identitäten auch in diesem Flow gestohlen werden können, liegt daran, dass er keine Lösung für das erstgenannte Problem liefert: zwischen App- und Web-Client existiert keine Kopplung. Jede Websession ist dazu

in der Lage, sich einzuloggen. Dabei ist der in Schritt 3 genutzte Parameter irrelevant, auch ein MAC-Token, wie in Abschnitt 2.3 erläutert, kann dieses Problem nicht lösen, denn jede beliebige Session wäre zu seiner, stets identischen, Berechnung imstande.

3.2 Gekoppelte Flows

Eine Möglichkeit, das Problem der Unabhängigkeit von Web-Session und App-Client aufzulösen, ist es, den Übermittelten Access Token oder Authorization Code asymmetrisch zu verschlüsseln. Da JWEs bereits in Kapitel 2.4 angesprochen wurden, sollen sie hier als Container für die Übermittlung eines Access-Tokens genutzt werden, auch wenn dir so entstehende Overhead fragwürdig ist, da nur ein einziger Parameter übermittelt wird. Der in Abbildung 3.3 konstruierte Flow kann jedoch auch mit anderen Container-Formaten genutzt werden.

(1) propose/redirect — Aufruf einer redirect-API, die als Parameter ausschließlich einen redirect-Key erhält. Im ersten Schritt stellt die Gekoppelte Flows fest, ob die genutzte Session bereits eingeloggt ist. Falls die Session eingeloggt ist, wird der angefragte Redirect ausgeführt —

(2–4) Key proposal — ist die Session jedoch ausgeloggt, so erstellt das Web-Backend ein neues asymmetrisches Schlüsselpaar. Der Private Key wird entweder direkt in der Session gespeichert oder in einer Datenbank mit der Session assoziiert und persistiert. Der Public Key muss in einer Datenbanktabelle gespeichert werden. Das ist notwendig, um in Schritt 5 zu verifizieren, dass der übermittelte Public Key auch tatsächlich im Backend erstellt wurde. Ohne diese Überprüfung könnten Drittanwendungen der App in Schritt 4 eigene, selbst generierte Public Keys übergeben. Abschließend wird der Public Key an den App-Client zurückgeleitet.

(5–6) Access-Token und JWE — Der App-Client nimmt den Public Key entgegen und leitet ihn an sein Backend weiter, indem er eine Protected API aufruft. Diese API überprüft im ersten Schritt, ob der Public Key in der Datenbank eingetragen ist. Anschließend kontaktiert sie den Authorization Server und lässt sich einen Access-Token ausstellen. Anschließend schreibt sie dieses Token in ein JWE, verschlüsselt und serialisiert es. Das so abgesicherte JWE wird an den App-Client redirectet.

(7–8) JWE Weiterleitung — Der App-Client leitet das JWE weiter an den User-Agent und dieser an das Web-Client-Backend. Dazu wird eine API aufgerufen und das serialisierte JWE als Query Parameter angefügt.

(9) JWE Entschlüsselung — Der in Schritt 2–4 in der Session bzw. einer Datenbank gespeicherte Private Key wird geladen. Anschließend wird das JWE entschlüsselt und der mitgelieferte Access-Token ausgelesen.

(10–11) User Identifizierung — Der in Schritt 9 ausgelesene Access-Token wird an den Authorization Server geleitet und von diesem verifiziert. Falls der Token valide ist, wird die User-ID mithilfe des Access-Tokens des Resource Owners ermittelt und dem Web-Client übergeben

(12–13) Login & Redirect — Anhand der User-ID wird die Session eingeloggt und an die ursprünglich angefragte Seite redirectet.

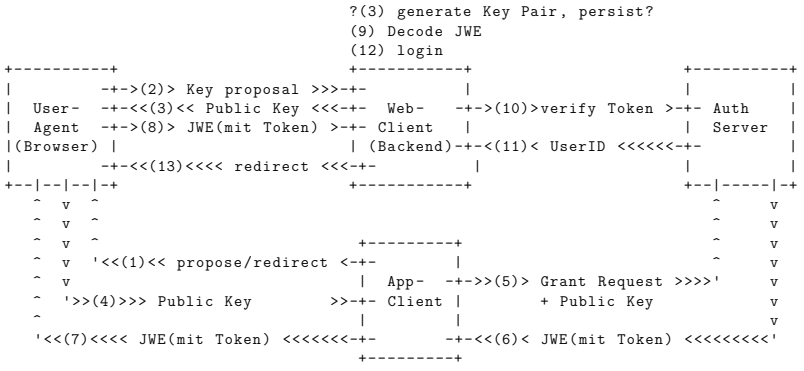

Abb. 3.3: Anwendungsfall, Implicit Login Flow

Kapitel 4

Fazit

Diese Arbeit gibt eine Übersicht, über die zwei grundlegenden OAuth Grant Types, Authorization Code Grant und Implicit Grant. Es wurde diskutiert, wie OAuth trotz unsichere User-Agents implementiert werden kann. Durch Betrachtung beispielhafter Flow-Diagramme wurde die Übermittlung von Tokens im Klartext ausgeschlossen. Aufgrund der besonderen Situation, dass die Websession zu Beginn nicht auf den Nutzer bezogen werden kann, wurden auch MAC-Tokens verworfen. Die Übermittlung wurde schließlich durch asymmetrisch verschlüsselte JWE Container gelöst. Von der Übermittlung eines unverschlüsselten Access-Keys ist in jedem Fall abzuraten. Eine Kopplung zwischen App- und Web-Client, wie in Kapitel 3.2 vorgeschlagen, ist zu empfehlen, da sie das Risiko, das von entwendeten Parametern ausgeht, minimiert. Eine Verbesserung würde die Erweiterung des vorgeschlagenen Flows (Abbildung 3.3) auf einen Authorization Code Grant bieten. Aufgrund der erhöhten Komplexität wurde davon in dieser Arbeit letztlich abgesehen.

Anhang A

Anhang

```
import com.nimbusds.jose.JOSEException;

import java.security.KeyPair;
import java.security.NoSuchAlgorithmException;
import java.text.ParseException;

public class Main {
    public static void main(String[] args) throws NoSuchAlgorithmException, JOSEException, ParseException {
        KeyGenerator keyGenerator = new KeyGenerator();
        JweGenerator jweGenerator = new JweGenerator();
        KeyPair keyPair = keyGenerator.generateKeyPair();
        String jweInText = jweGenerator.buildEncryptedCompactJWT(keyPair.getPublic());
        jweGenerator.decryptJWT(jweInText, keyPair.getPrivate());
    }
}
```

Abb. A.1: main-Methode zur ver- und entschlüsselung eines JWEs in Java mit nimbusds

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;

public class KeyGenerator {
    public KeyPair generateKeyPair() throws NoSuchAlgorithmException {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
        return keyPairGenerator.genKeyPair();
    }
}
```

Abb. A.2: Asymmetrischer Key Generator, RSA

```
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.UUID;

import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.RSADecrypter;
import com.nimbusds.jose.crypto.RSAEncrypter;
import com.nimbusds.jwt.EncryptedJWT;
import com.nimbusds.jwt.JWTClaimsSet;

public class jweGenerator {

    public String buildEncryptedJWT(PublicKey publicKey) throws JOSEException {

    public String buildEncryptedCompactJWT(PublicKey publicKey) throws JOSEException {

        JWTClaimsSet jwtClaims = new JWTClaimsSet();
        jwtClaims.setCustomClaim("token","mytoken");
        jwtClaims.setExpirationTime(new Date(new Date().getTime()-600));
        Date currentTime = new Date();
        jwtClaims.setNotBeforeTime(currentTime);
        jwtClaims.setIssueTime(currentTime);
        jwtClaims.setJWTID(UUID.randomUUID().toString());

        JWEHeader jweHeader = new JWEHeader(JWEAlgorithm.RSA_OAEP, EncryptionMethod.A128GCM);
        JWEEncrypter encrypter = new RSAEncrypter((RSAPublicKey) publicKey);
        EncryptedJWT encryptedJWT = new EncryptedJWT(jweHeader, jwtClaims);
        encryptedJWT.encrypt(encrypter);

        String joseHeader = encryptedJWT.getHeader().toBase64URL().toString();
        String encKey = encryptedJWT.getEncryptedKey().toString();
        String initVect = encryptedJWT.getInitializationVector().toString();
        String ciph = encryptedJWT.getCipherText().toString();
        String tag = encryptedJWT.getIntegrityValue().toString();
        String jwtInText = joseHeader +"."+encKey+"."+initVect+"."+ciph+"." + tag;
        System.out.println(jwtInText);

        return jwtInText;
    }
}
```

Abb. A.3: Klasse zur ver- und entschlüsselung eines JWE-Tokens mit nimbusds

```
  public void decryptJWT(String jwtInText, PrivateKey privateKey) throws ParseException, JOSEException {
      JWEDecrypter decrypter = new RSADecrypter((RSAPrivateKey) privateKey);
      EncryptedJWT encryptedJWT = EncryptedJWT.parse(jwtInText);
      //encryptedJWT.decrypt(decrypter);
      System.out.println("JWE Header:" + encryptedJWT.getHeader());
      System.out.println("JWE Content Encryption Key:" + encryptedJWT.getEncryptedKey());
      System.out.println("Initialization Vector:" + encryptedJWT.getInitializationVector());
      System.out.println("Ciphertext:" + encryptedJWT.getCipherText());
      System.out.println("Payload:" + encryptedJWT.getPayload());
    }
}
```

Abb. A.4: Klasse zur ver- und entschlüsselung eines JWE-Tokens mit nimbusds

Abbildungsverzeichnis

Literaturverzeichnis

Chari, S., Jutla, C. S. & Roy, A. (2011), 'Universally composable security analysis of oauth v2.0', IACR Cryptology ePrint Archive 2011, 526.
URL: http://eprint.iacr.org/2011/526

Hammer, E. (2012), 'Oauth 2.0 and the road to hell'.

Hardt, D. (2012), The OAuth 2.0 Authorization Framework, number 6749.
URL: https://tools.ietf.org/html/rfc6749

Hu, P., Yang, R., Li, Y. & Lau, W. C. (2014), Application impersonation: Problems of oauth and api design in online social networks, in 'Proceedings of the Second ACM Conference on Online Social Networks', COSN '14, ACM, New York, NY, USA, pp. 271–278.
URL: http://doi.acm.org/10.1145/2660460.2660463

Petrlic, R. & Sorge, C. (2017), Datenschutz - Einführung in technischen Datenschutz, Datenschutzrecht und angewandte Kryptographie, Springer Vieweg, Wiesbaden.

Rohr, M. (2018), Sicherheit von Webanwendungen in der Praxis, 2 edn, Springer Vieweg, Wiesbaden.

Sakimura, N., J., B., M., J., de Medeiros B. & C., M. (2014), Openid connect core 1.0, Technical report, OpenID Foundation.

Siriwardena, P. (2014), Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE, 1st edn, Apress, Berkely, CA, USA.

Sun, S.-T. & Beznosov, K. (2012), The devil is in the (implementation) details: An empirical analysis of oauth sso systems, in 'Proceedings of the 2012 ACM Conference on Computer and Communications Security', CCS '12, ACM, pp. 378–390.
URL: http://doi.acm.org/10.1145/2382196.2382238

Yang, R., Li, G., Lau, W. C., Zhang, K. & Hu, P. (2016), Model-based security testing: An empirical study on oauth 2.0 implementations, in 'Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security', ASIA CCS '16, ACM, pp. 651–662.
URL: http://doi.acm.org/10.1145/2897845.2897874