Web-Anwendungen: REST-Security

Web-Engineering II

Prof. Dr. Sebastian von Klinski

IT-Security bei REST-Services

- Bei REST-Services müssen in der Regel die folgenden Aspekte zum Schutz der Daten und User umgesetzt werden

  - **Authentifizierung**: Prüfung, ob der Nutzer der ist, der er vorgibt zu sein
  - **Autorisierung**: Prüfung, ob der Nutzer die angefragten Aktionen ausführen darf
  - **Vertraulichkeit**: Keine unberechtigten Personen können die Nachrichten zwischen Client und Server lesen.
  - **Integritätsprüfung**: Die Nachrichten zwischen Client und Server können nicht unbemerkt geändert werden.

Authentifizierung

Authentifizierung

- Verfahren zur Bestätigung, dass ein Nutzer der ist, der er vorgibt zu sein
- In der Regel umgesetzt durch **User-ID** und **Passwort**
- Durch die Eingabe des richtigen Passworts belegt der Nutzer, dass er berechtigt ist, den Account zu verwenden
- Voraussetzung hierfür ist
  - die sichere Verwahrung der User-Daten
  - der sichere Austausch der Zugangsdaten im Prozess der Authentifizierung (siehe Vertraulichkeit)
  - der Schutz der Passwörter für den Fall, dass die User-Daten in unberechtigte Hände gelangen

User-Daten

- Für die Authentifizierung gibt es in den meisten Systemen eine Tabelle mit Usern
- Ein User beinhaltet in der Regel die folgenden Daten
  - **UserID**
    - eindeutiger Name
    - in vielen B2C-Anwendungen heute die Email-Adresse
    - Email-Adresse ist zwar lang, aber bei vergessenen Passwort ist das Zusenden des neuen Passworts sehr einfach
  - **Passwort**
    - der geheime Schlüssel zur Authentifizierung
  - **Angaben zur natürlichen Person** (z.B. Vor- und Nachname)

Beim Ablegen von User-Daten sind einige Aspekte zu beachten…

User-Daten

- **Passwörter dürfen nie im Klartext abgelegt werden**
  - Auch Systemadministratoren/ Software-Entwicklern kann man nicht immer trauen
    - Mehr als 50% aller IT-Angriffe in Organisationen werden durch interne Mitarbeiter/innen ausgeführt!
  - Auch die Passwörter zu Datenbanken sind nicht immer sicher
  - Datenbanken können gehackt werden
  - Jedem, auch Administratoren passieren Fehler
  - Viele Nutzer benutzen Passwörter mehrfach →
    - wird ein System kompromittiert, in dem das Passwort lesbar ist, dann sind auch andere Systeme in Gefahr
- Aus diesen und anderen Gründen dürfen Passwörter nie im Klartext in der Datenbank abgelegt werden!

Passwörter speichern

- In der Regel wird ein Hashwert vom Passwort in der Datenbank abgelegt
- Hashing
  - Hashing ist das Anwenden einer Hashfunktion auf das Passwort
  - Eine Hashfunktion (auch Streuwertfunktion) ist eine unidirektionale Abbildung von einem Eingangswert (z.B. Klartextpasswort) in einen Zielwert (gehashtes Passwort).
- Hashfunktionen haben die Eigenschaften
  - Der Zielwert ist bei gleichem Eingangswert immer gleich
  - Aus dem Zielwert kann der Eingangswert nicht wiederhergestellt werden
  - Es ist fast unmöglich, zwei Eingangswerte zu finden, die den gleichen Zielwert haben
- Beispiel
  - Passwort: Dasistgeheim
  - Hashwert: QWxhZGRpbjpPcGVuU2VzYW1l

Passwörter speichern

- Gängige Hash-Funktionen
  - SHA (Secure Hash Algorithm)
    - Gruppe standardisierter kryptologischer Hashfunktionen
    - Dienen zur Berechnung eines Prüfwerts für beliebige digitale Daten (Nachrichten) und sind unter anderem die Grundlage zur Erstellung einer digitalen Signatur
    - Varianten SHA-1, SHA-2 und SHA-3
  - Blowfish
    - Blockverschlüsselungsalgorithmus, der 1993 von Bruce Schneier entworfen wurde
    - Unpatentiert und gemeinfrei veröffentlicht (Bruce Schneier beansprucht nicht das Urheberrecht!)
    - Kann frei verwendet werden
    - Feste Blocklänge von 64 Bit

Quelle: https://de.wikipedia.org/wiki/Secure_Hash_Algorithm

Ablauf der Authentifizierung

- In der Datenbank ist der Hashwert des Passwortes abgelegt.
- Der Benutzer gibt das Passwort zur Identifizierung ein.
- Es wird auf das eingegebene Passwort die gleiche Hashfunktion angewandt, mit der das Passwort des Users gespeichert wurde.
- Die Hashwerte des eingegebenen und gespeicherten Passworts werden verglichen.
- Wenn beide Hashwerte gleich sind, hat der Benutzer das richtige Passwort eingegeben.

Doch dieser Schutz der Passwörter über eine Hashfunktion reicht nicht!

Quelle: https://de.wikipedia.org/wiki/Secure_Hash_Algorithm

Angriffe auf Hashwerte

- Häufiges Ziel von Hackerangriffen: User-Daten in angegriffenen Systemen
- Wenn diese Daten erbeutet wurden, geht es an das Hacken der Passwörter
- **Wörterbuchangriff**
  - Ein Wörterbuch ist eine Sammlung von gängigen Passwörtern
  - Beispielsweise
    - Spaßpasswörter oder leicht einzugebende Passwörter: asdf
    - Geburtsdaten: 23.06.1991
    - Namen: Monika
    - Kombinationen von diesen
  - Da begrenzte Anzahl von Hash-Funktionen verfügbar, können Hacker für gängige Passwörter den Hashwert berechnen und mit den Werten in den erbeuteten Tabellen vergleichen
  - Wenn Nutzer ein gängiges Passwort verwendet haben, ist es nur eine Frage der Zeit, bis das Passwort geknackt ist.

Rainbow-Tables

- Rainbow-Tables sind Tabellen mit vielen hundert Millionen Passwörtern, für die bereits die Hashwerte für verschiedene Hashfunktionen berechnet wurden
- Während die Berechnung von Hashwerten vergleichsweise lange dauert, ist der Vergleich von Werten sehr schnell
- Durch die Verwendung von Rainbow-Tables können in sehr kurzer Zeit viele mögliche Passwörter ausgetestet werden

Verwendung von Salt

- Sowohl Wörterbuchangriffe als auch Rainbow-Table-Angriffe können durch das Verwenden von Salts effektiv erschwert/ verhindert werden
- **Salt**
  - Beim Ablegen des Passworts wird die Hashfunktion nicht nur auf das Passwort angewandt.
  - Das Passwort wird verlängert um einen weiteren Text (der Salt).
  - Der Salt ist ein der Regel ein längerer Zufallstext
  - Dieser Text ist entweder für alle Nutzer gleich oder für jeden Nutzer anders.
  - Dieser ergänzende Text kann auch bekannt sein.

Verwendung von Salt

- Durch das Verwenden von Salts…
  - … funktionieren Wörterbuchangriffe nicht mehr, weil das Passwort nicht einem gängigen Passwort gleicht
  - … funktionieren Rainbow-Tables nicht mehr, weil für jedes System, ggfls. für jeden Nutzer der Salt anders ist
- Durch die Verwendung von Salts bliebt nur noch der Brute-Force Angriff
- Es müssen alle möglichen Hashwerte berechnet werden
- Das ist heutzutage für eine gute Hashfunktion selbst mit den schnellsten Rechnern nicht sinnvoll

Passwort: Dasistgeheim

Ursprünglicher Hashwert: QWxhZGRpbjpPcGVuU2VzYW1l

Salt: $2b$10$KHEom97oR5Fug65CbXe6.u

Passwort mit Salt: Dasistgeheim$2b$10$KHEom97oR5Fug65CbXe6.u

Hash:

$2b$10$KHEom97oR5Fug65CbXe6.uPhj6U8ER26Ivs2kgXpRiGK040O9eNJO

Bcrypt

- Mit dem Modul bcrypt können Hashwerte mit Salt berechnet werden
- **node.bcypt.js**
  - bcrypt ist eine kryptologische Hashfunktion speziell für das Hashen und Speichern von Passwörtern basierend auf Blowfish-Algorithmus
- Installation: ***npm install bcrypt***
- Vorgehensweisen
  - Option 1: Salt generieren, dann Hash mit Salt berechnen
  - Option 2: Beide Schritte in einem ausführen
- Das Ergebnis ist das gleiche

Hashing mit Salt

```
const bcrypt = require("bcrypt");
const saltRounds = 10;
const plainTextPassword1 = "DFGh5546*%^__90";

bcrypt
    .genSalt(saltRounds)
    .then(salt => {
        console.log(`Salt: ${salt}`);
        return bcrypt.hash(plainTextPassword1, salt);
    })
    .then(hash => {
        console.log(`Hash: ${hash}`);
        // Store hash in your password DB.
    })
    .catch(err => console.error(err.message));
```

Salt generieren

Hash erzeugen

Hashing mit Salt

- Ergebnis

```
PS C:\Git\de.beuth.svk.lehre.we2.rest-server01\de.beuth.svk.lehre.we2.rest-server01> node .\testing\bcrypt-test.js

Salt:
$2b$10$KHEom97oR5Fug65CbXe6.u

Hash:
$2b$10$KHEom97oR5Fug65CbXe6.uPhj6U8ER26Ivs2kgXpRiGK040O9eNJO
```

- Hash umfasst Salt und Passwort in einem String

Nutzung von Bcrypt mit Mongoose

- Anwenden der Hashfunktion kann gleich im Objektmodell der Klasse User verankert werden

```
var mongoose = require('mongoose');
var bcrypt = require('bcryptjs')
var userSchema = new mongoose.Schema({
    first: String,
    email: {type:String,unique:true},
    password: String,
    image:String
},{timestamps:true})
…
```

Nutzung von Bcrypt mit Mongoose

- Im Schema können mit pre() Aktionen registriert werden, die ausgeführt werden, bevor in diesem Fall ein „save“ ausgeführt wird

```
…
userSchema.pre('save', function (next) {
    var user = this;
    if (!user.isModified('password')) {return next()};
    bcrypt.hash(user.password,10).then((hashedPassword) => {
        user.password = hashedPassword;
        next();
    })
}, function (err) {
    next(err)
})
…
```

Nutzung von Bcrypt mit Mongoose

- Auch die Methode zum Vergleich von eingegebenen und gespeicherten Passwort kann gleich im Schema hinterlegt werden.
  - Alle Aspekte des Kodierens und Vergleichens des Passworts an einer Stelle
  - Saubere Modularisierung fördert die Konsistenz der Daten

```
…
userSchema.methods.comparePassword=function(candidatePassword,next){
bcrypt.compare(candidatePassword,this.password,function(err,isMatch){
        if(err)
                return next(err);
        next(null,isMatch)
    })
}
module.exports = mongoose.model("user", userSchema);
```

Authentifizierung bei REST-Services

Authentifizierung von REST-Services

- Für die Authentifizierung bei REST-Services muss zunächst festgelegt werden, wie die Zugangsdaten sicher übertragen werden können
- Standardisierter Ansatz für Übertragung wäre vorteilhaft → ansonsten ist Wiederverwendung beeinträchtigt

- Mögliche Optionen für das Übertragen von User-ID und Passwort
  - HTTP/S: Basic Access Authentication
    - Standardzugriffsauthentifizierung von HTTP
  - Über entsprechende Felder in den übertragenen Daten

HTTP Basic Authentication

- Basierend auf HTTP
- Request-Header hat Feld: **Authorization: Basic**
- Zugangsdaten werden Base64 encoded
  - Serialisierung, keine Verschlüsselung!
- User-ID und Passwort sind durch Doppelpunkt getrennt
  - Aladdin:OpenSesame → QWxhZGRpbjpPcGVuU2VzYW1l
  - Im Header: ***Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l***
- Vorteile
  - Benötigt keine Cookies, Session Identifiers, Login-Seiten oder ähnliches
  - Benutzt ausschließlich Standardfelder des HTTP-Headers
- Aber
  - Basic Authentication gewährt keine Vertraulichkeit (confidentiality)
  - Daher sollte es immer in Verbindung mit HTTPS genutzt werden

Basic Authentication

- Protokollschritte
  1. Client fragt URL ab
  2. Server antwortet mit “HTTP 401 Unauthorized status” und einem WWW-Authenticate-Feld

```
WWW-Authenticate: Basic realm="User Visible Realm", charset="UTF-8"
```

3. Client antwortet mit Request, bei dem Zugangsdaten enthalten sind

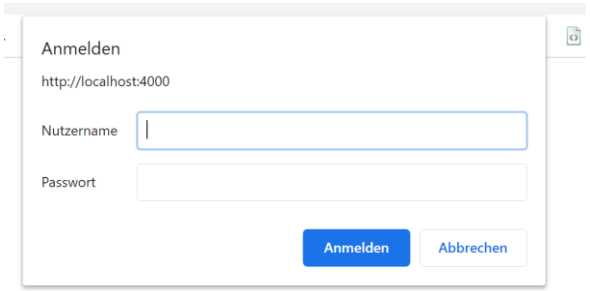
```
post http://localhost:3000/messages/
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
Content-Type: application/json
{
   "headline": "Das ist ganz neu",
   "messageText": "Morgen ist Weihnachten"
}
```

Basic Authentication in Node.js

- Basic Authentication kann durch Abfragen der HTTP-Header-Daten problemlos umgesetzt werden
- Schritt 1: Prüfen, ob Angaben im Header sind, falls nicht, Daten anfordern

```
if (!req.headers.authorization || req.headers.authorization.indexOf('Basic ') === -1) {
        res.statusCode = 401;
        res.setHeader('WWW-Authenticate', 'Basic realm="Secure Area"');
        return res.json({ message: 'Missing Authorization Header Gib die daten' });
    }
```

- Durch Setzen von res.setHeader(,WWW-Authenticate… wird Browser aufgefordert, Login-Fenster zu zeigen



Basic Authentication in Node.js

- Wenn anstelle des Browsers ein REST-Call erstellt wird, muss in der Request-Message der entsprechende Header gesetzt werden

```
http://localhost:4000/
Authorization: Basic dGVzdDp0ZXN0
```

- dGVzdDp0ZXN0 ist [UserID]:[Passwort] in Base64 encoding
- Erstellen des Calls mit Axios

```
const axios = require('axios');
axios.get('http://localhost:4000')
  .then(response => {
    console.log(response.data.url);
})
  .catch(error => {
    var authenticationValue = error.response.headers['www-authenticate'];
    console.log("Authenticate? " + authenticationValue);
});
```

Basic Authentication in Node.js

- Schritt 2: Wenn die Zugangsdaten im Header sind,
  - werden sie ausgelesen,
  - aus dem Base64-Code muss der Klartext erstellt werden,
  - User-ID und Passwort, durch „:“ getrennt werden und
  - mit den User-Daten abgeglichen werden

```
const base64Credentials = req.headers.authorization.split(' ')[1];
    const credentials = Buffer.from(base64Credentials, 'base64').toString('ascii');
    const [username, password] = credentials.split(':');
    const user = await userService.authenticate({ username, password });
    if (!user) {
        return res.status(401).json({ message: 'Invalid Authentication Credentials' });
    }
```

Authentifizierung: In JSON-Nachricht

- Alternativ können Zugangsdaten auch mit der Nachricht übertragen werden
- Bei Datenformaten wie XML und JSON können die Zugangsdaten problemlos bei jeder Anfrage integriert werden

```
post http://localhost:3000/messages/
Content-Type: application/json
{
    "Request": "Login",
    "UserID": "Max",
    "Password": "Sesam",
}
```

- Nachteile
  - GET-Methoden übertragen eigentlich keinen Content → wären nicht nutzbar für GET-REST-Services
  - Es müsste auch HTTPS verwendet oder die Nachricht verschlüsselt werden

Access-Tokens und JWT

Authentifizierung von REST-Services

- Nutzung von REST-Services ist durch folgende Aspekte geprägt
  - Die Clients greifen selbst innerhalb einer Clients-Session sehr oft auf den Service zu
  - REST-Services sollten zustandslos sein
- Problemstellung
  - Wenn der Service zustandslos ist, muss eigentlich mit jedem Aufruf eine Authentifizierung durchgeführt werden

HTTP GET
`/api/users/33245/messages`

Gib mir alle Messages des Nutzers 33245

HTTP DELETE
`/api/message/332456`

Lösche Message 332456

Authentifizierung von REST-Services

- Wenn bei jeder Anfrage eine Authentifizierung vorgenommen werden müsste, müssten die Zugangsdaten mit jeder Anfrage verschickt werden
- Permanentes Versenden der Zugangsdaten ist problematisch
  - Die Wahrscheinlichkeit, dass die Daten abgegriffen und gehackt werden, steigt deutlich an.
  - Die Zugangsdaten, insbesondere das Passwort im Klartext muss permanent im Client vorgehalten werden
    - Wenn der Client gehackt wird, ist das Passwort zugänglich
  - Passwörter im Klartext sollten nie lange in einer Applikation vorhanden sein!
- Ein Vorhalten und wiederholtes Versenden der Zugangsdaten sollte vermieden werden
- Die gängige Lösung hierfür: Access Tokens

Access-Tokens

- **Access-Tokens** identifizieren Sicherheitsanmeldeinformationen für eine Anmeldesitzung und erlauben die Identifizierung oder Berechtigung eines Benutzers
- Access-Tokens vermeiden das ständige Versenden der Zugangsdaten
  - Client authentifiziert sich gegenüber dem Server
  - Server gibt dem Client einen Access-Token zurück
  - Bei den kommenden Requests gibt der Client nur noch den Token mit, anhand dessen der Server feststellen kann, dass der User berechtigt ist, den Service zu nutzen
- Token hat in der Regel eine TTL (Time to live)
- Nach Ablauf der TTL ist der Token nicht mehr gültig und ein neuer muß angefordert werden

Access-Tokens: Varianten

- **Zustandsbehaftet**:
  - Server kennt Tokens und kann mit Token User und Rechterolle verbinden
  - **Vorteil**: detaillierte Autorisierung ist möglich
  - **Nachteil**: Service ist nicht mehr zustandslos, keine einfache Skalierung

- **Zustandslos**: Token bestätigt pauschal, dass User den Service nutzen kann
  - **Vorteil**: zustandsloser Service kann problemlos skaliert werden
  - **Nachteil**:
    - Entweder ist keine detaillierte Autorisierung möglich, weil nicht mehr festgestellt werden kann, wer der betreffende User ist und welche Rechterollen er hat
    - Oder es müssten rechtebezogene Daten per Token ausgetauscht werden (nicht ratsam, aber möglich/ gängig)

JWT (JSON Web Token)

- Bibliothek für Access-Tokens basierend auf JSON und nach RFC 7519
  - Zustandslose Tokens
  - Austausch von verifizierbaren Claims (nicht nur Login-Daten)
- Üblicher Anwendungsfall
  - Nutzer fragt Zugriff auf Ressource (z.B. URL) bei **Service-Provider** an
  - Ein **Identity-Provider** prüft Identität (Authentifizierung) und vergibt Token, wenn Zugriff zulässig
  - **Service-Provider** prüft nur noch Gültigkeit des Tokens
- Tokens können signiert und verifiziert werden, um vor Veränderung zu schützen

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6InNjaXAgQUciLCJhZG1pbiI6dHJ1ZSwiaWF0IjoxNTE2MjM5MDIyfQ.dX_ACtDZVh9ZKaDMLmZFURrh7o5R99-6MEulWUFCaT0

JWT (JSON Web Token)

- Grundlegender Aufbau
  - **Header:**
    {"alg": "HS256", "typ": "JWT"}
  - **Payload:**
    {"sub": "1234567890", "name": "scip AG", "admin": true, "iat": 1516239022}
  - **Signatur mit HS256:**
    dX_ACtDZVh9ZKaDMLmZFURrh7o5R99-6MEulWUFCaT0
- Durch Punkte „.“ getrennt

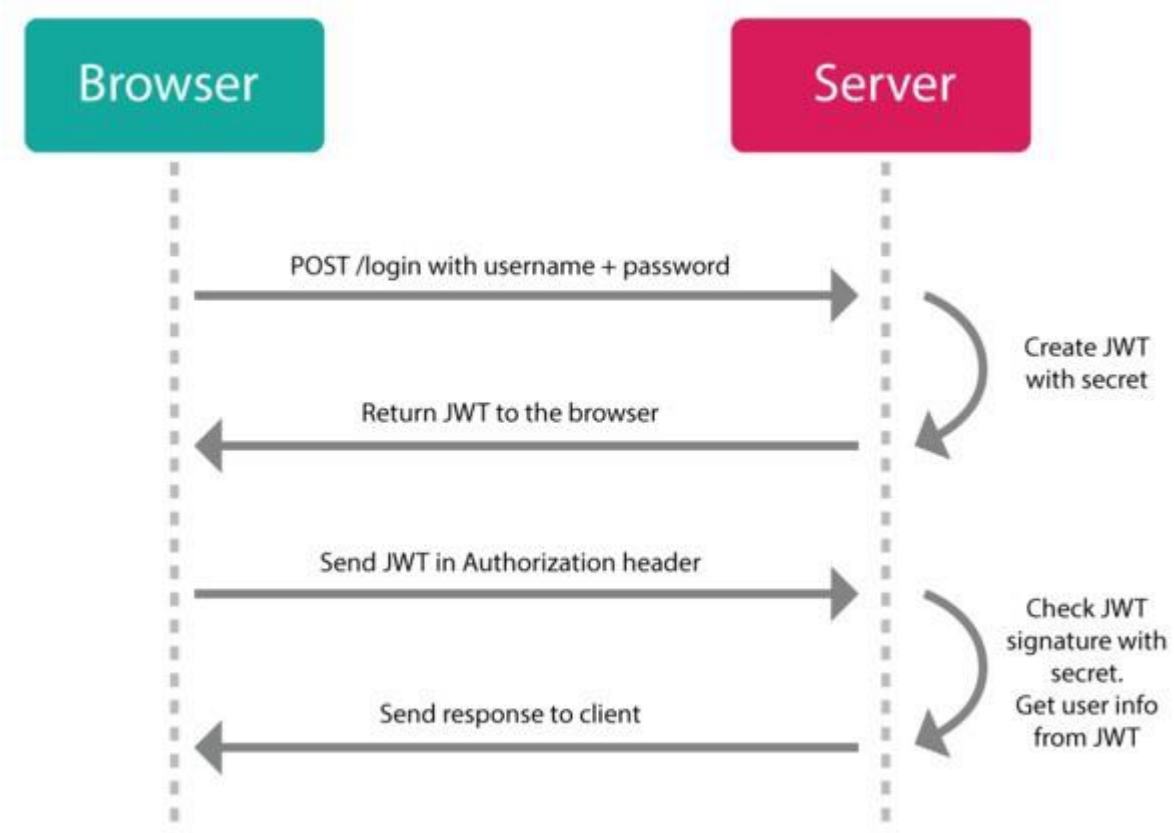
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6InNjaXAgQUciLCJhZG1pbiI6dHJ1ZSwiaWF0IjoxNTE2MjM5MDIyfQ.dX_ACtDZVh9ZKaDMLmZFURrh7o5R99-6MEulWUFCaT0

JWT (JSON Web Token)

- Header
  - Gibt Hash-Funktion für Signatur an
  - Kann aber auch ohne Algorithmus sein: {"alg":"none"} (nicht sicher!)
  - Gängig ist HS256
- Payload
  - JSON-Objekt
  - Nicht verschlüsselt, sondern nur Base64-URL-encoded!
  - **Jeder kann Token lesen!**
- Signatur
  - Sollte nicht zu schwach sein (Secret Key)
  - Ansonsten kann man Token verändern

Access-Tokens: JWT (JSON Web Token)

- Ablauf der Kommunikation



Quelle: https://medium.com/swlh/a-practical-guide-for-jwt-authentication-using-nodejs-and-express-d48369e7e6d4

JWT (JSON Web Token)

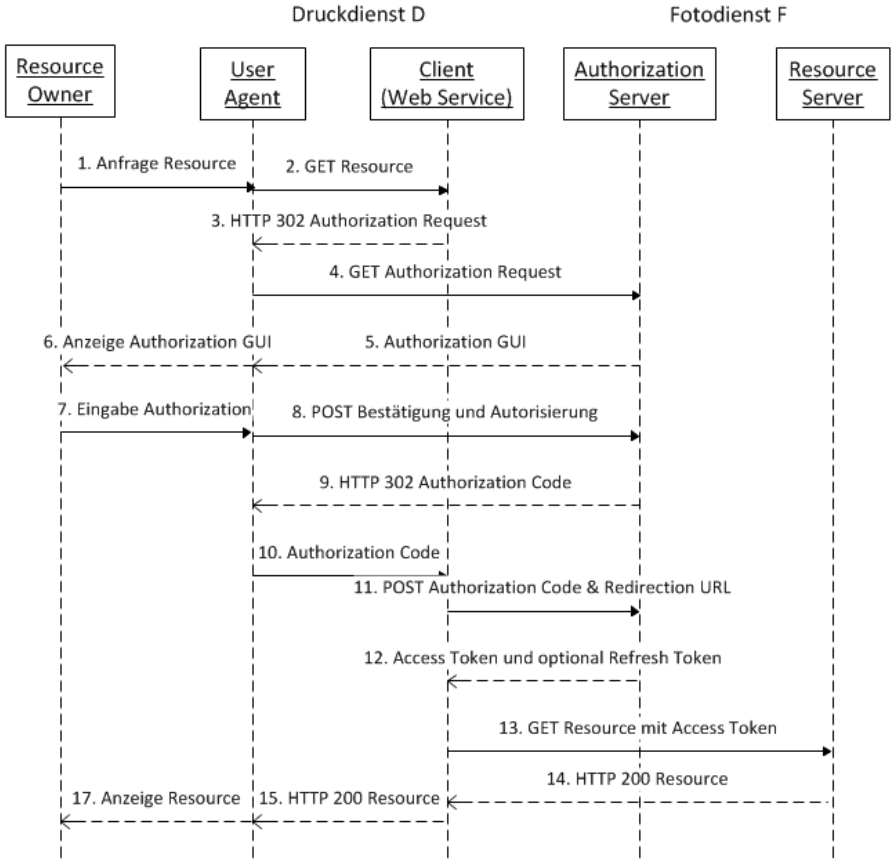
- Da Token-Payload gelesen werden kann, sollten keine sicherheitskritischen Angaben (z.B. User-ID, Rolle, etc.) unverschlüsselt übertragen werden
- Entweder Verschlüsselung des Tokens oder der gesamten Kommunikation (z.B. HTTPS) dringend notwendig
- Sicherheit kann erhöht werden, indem im Payload die Client-Adresse mitgeliefert wird → Übernehmen des Tokens durch andere erschwert
- Refresh-Tokens
  - Üblicherweise wird TTL von Access-Tokens eher kurz gehalten
  - Um Sitzung fortzuführen werden Refresh-Tokens vergeben
  - Ausgabe von Refresh-Tokens muss sicher sein, da sonst unbefugte „Token-Sessions“ übernehmen können!

Access-Tokens: JWT (JSON Web Token)

- Durch…
  - … die Trennung von autorisierende Stelle und Ressource-Provider sowie
  - durch die Validierbarkeit von den Tokens
- … kann die Freigabe von Ressourcen (Services, Daten, Bilder, etc.) sehr flexibel gestaltet werden
  - Der Aussteller des Tokens muss nicht auch der Service Provider sein, auf den der Client zugreifen will
- Ansatz von OAuth, um flexibel Services und Ressourcen im Internet freizugeben
- Beispiel: Freigabe von Daten
  - Nutzer A will auf Daten in Cloud von Nutzer B zugreifen
  - Nutzer B stellt für Nutzer A einen zeitlich befristetes Token für die konkreten Daten aus
  - Nutzer A kann mit Token auf Daten in der Cloud zugreifen
  - **Es mussten keine Zugangsdaten verschickt werden!**

OAuth

- Token-basierte standardisierte Autorisierung, die von mehreren Internetdiensten (z.B. Google) unterstützt wird
- Ansatz: über Access-Tokens kann die Berechtigung für die Nutzung von Ressourcen vergeben werden, ohne dass dazu Zugangsdaten verschickt werden müssen
- JWT kann für diese Art der Autorisierung verwendet werden



Quelle: Wikipedia

JWT in Node.js

- JWT auch als Modul für Node.js verfübar
- Installation: **npm install jsonwebtoken**
- Für Nutzung von JWT sind 3 Aktionen umzusetzen
  - Token anlegen, sofern die übertragenen Zugangsdaten richtig sind und an den Client zurücksenden
  - Token überprüfen bei allen Anfragen (als Middleware, damit Token bei allen Anfragen geprüft wird)
  - Token erneuern, falls TTL bald am Ende

JWT in Node.js

- Anlegen von Token

```
const jwt = require('jsonwebtoken')
…
const signIn = (req, res) => {
  // Determine User-ID and Password from Request, verify and if it is ok…
  …
  // User-ID and Password are correct. Can create token
  // Private Key for signing the token
  const jwtKey = 'my_secret_key‘;
  // TTL
  const jwtExpirySeconds = 300;

  const token = jwt.sign({ username }, jwtKey, {
    algorithm: 'HS256',
    expiresIn: jwtExpirySeconds
  })
  // Return token to client
```

JWT in Node.js

- Verifizieren von Token wird üblicherweise als “Middleware” umgesetzt (auf alle Anfragen angewandt)

```
…
 // Retrieve the token from the header or as a cookie
 …
 // If there is a token, verify it the with secret key
 try {
   var payload = jwt.verify(token, jwtKey)
   } catch (e) {
      // if the token is wrong, an exception is thrown
      if (e instanceof jwt.JsonWebTokenError) {
         // Not logged in, redirect to error page
      }
   // Do the action the user requested
  …
```

JWT in Node.js

- Token erneuern, bevor the TTL abgelaufen ist

```
…
var payload
  try {
    payload = jwt.verify(token, jwtKey)
  } catch (e) {
    if (e instanceof jwt.JsonWebTokenError) {
      return res.status(401).end()
    }
    return res.status(400).end()
  }
  const nowUnixSeconds = Math.round(Number(new Date()) / 1000)
  if (payload.exp - nowUnixSeconds > 30) {
    return res.status(400).end()
  }
  const newToken = jwt.sign({ username: payload.username }, jwtKey, {
    algorithm: 'HS256',
    expiresIn: jwtExpirySeconds
  })
…
```

Übertragen des Tokens

- Mehrere Optionen für das Übertragen des Tokens möglich
  - **In der URL** (für REST-Services nicht sinnvoll)

```
http://example.com/path?jwt_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

- Als Teil des **HTTP-Headers**
- Im Authorization-Feld als Bearer-Token

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

- Als **Cookie**

```
Cookie: token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..
```

- Jeder Ansatz hat seine Vor- und Nachteile

Authentifizierung: Ablauf

- Schützen aller Endpoints über eine Middleware
  - Callback versucht den Token zu lesen
  - Falls keiner vorhanden ist oder er nicht mehr gültig ist, Fehlermeldung zurückgeben (kein Zugriff auf die Ressource)
  - Falls der Token ok ist, mit next() die gewünscht Aktion durchführen
- Der Token sollten im HTTP-Header übertragen werden
- Standard-Header-Feld
  - Authorization: Bearer [Token]

Authentifizierung mit JWT

- Requests nach der Authentifizierung mit Token

```
…
http://localhost:3000/users
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJib2R5Ijoic3R1ZmYiLCJpYXQiOjE1ODM1MD
Y2ODR9.dkuUTWIUi4jUxQiwAa6qrsqcyaZx9UMueGhztF3qY1I…
```

- Prüfen des Tokens über den ersten von zwei Request-Handler

```
…
router.get('/', isAuthenticated, function (req, res, next) {
  userService.getUsers(function (err, result) {
    if (result) {
      res.send(Object.values(result));
      console.log(result);
    }
    …
```

Authentifizierung mit JWT

- Funktion zum Prüfen des Tokens

```
…
function isAuthenticated(req, res, next) {
  if (typeof req.headers.authorization !== "undefined") {
     let token = req.headers.authorization.split(" ")[1];
     var privateKey = config.get('session.tokenKey');
     jwt.verify(token, privateKey, { algorithm: "HS256" }, (err, user) => {
     if (err) {
        res.status(500).json({ error: "Not Authorized" });
        return;
      }
      return next();
    });
  } else {
    res.status(500).json({ error: "Not Authorized" });
    return;
  }
}
…
```

Vertraulichkeit

Vertraulichkeit

- Vertraulichkeit wird bei einer Kommunikation zwischen Client und Server durch Verschlüsselung erreicht
- Ansätze für Verschlüsselung
  - Symmetrische Verschlüsselung
    - AES (Advanced Encryption Standard)
    - Triple-DES: eine Weiterentwicklung des DES-Verfahrens
    - Blowfish, etc.
  - Asymmetrische Verschlüsselung
    - RSA,
    - Diffie-Hellman, etc.

Vertraulichkeit

- Symmetrische Verschlüsselung
  - Es wird ein geheimer Schlüssel ausgetauscht
  - Mit dem geheimen Schlüssel werden die Nachrichten ver- und entschlüsselt
  - Vorteil: sehr schnell
  - Nachteil: es muss erst ein Schlüssel ausgetauscht werden (Angriffsmöglichkeit)
- Asymmetrische Verschlüsselung
  - Für Verschlüsselung gibt es privaten und öffentlichen Schlüssel
  - Nachrichten können mit öffentlichen Schlüssel verschlüsselt werden
  - … aber nur mit privaten Schlüssel entschlüsselt werden
  - Vorteil: kein Austausch von Schlüssel notwendig
  - Nachteil: sehr langsam, da rechenaufwändig → nicht geeignet für große Datenvolumina bei Streams, Bildern, etc.

HTTPS

- HTTPS kombiniert Vorteile von symmetrischer und asymmetrischer Verschlüsselung
  - Der Schlüsselaustausch wird sicher über asymmetrische Verschlüsselung umgesetzt → sicher
  - Anschließend kann mit ausgetauschten Schlüssel die symmetrische Verschlüsselung für die weitere Kommunikation verwendet werden
- Auch bei großen Datenvolumina performant (z.B. Bilder und Videos)
- Damit Schlüsselaustausch möglich ist, muss Server privaten und öffentlichen Schlüssel haben → Server-Zertifikat notwendig

Vorgehensweise für Erstellung von Server-Zertifikat

- Zertifikat von Zertifizierungsstelle ausstellen lassen (in der Regel kostenpflichtig)
- Eigene Zertifikate erstellen
  - Etwas aufwändiger
  - Zertifikate werden von Browsern als nicht gültig markiert
  - HTTPS funktioniert aber nach Bestätigung
- Let‘s Encrypt
  - Kostenlose Zertifikate (in Node.js z.B. über Greenlock-Modul)
  - Einfache Umsetzung, automatische Verlängerung, etc.
  - Funktioniert jedoch nicht für „localhost“

HTTPS

- HTTPS für praktische alle REST-Services unverzichtbar
- Insbesondere, wenn
  - Es vertrauliche Daten gibt und
  - Es ein Login gibt
- Für Entwicklung sollte zunächst eigenes Server-Zertifikat erstellt werden

Eigene Zertifikate ausstellen

- OpenSSL installieren (falls nicht vorhanden)
- Schlüssel und Zertifikate erstellen

```
openssl req -x509 -newkey rsa:2048 -keyout keytmp.pem -out cert.pem -days 365
```

- PEM-Dateien erstellen (entschlüsselt)

```
openssl rsa -in keytmp.pem -out key.pem
```

Starten von Server mit Zertifikat

- Warnung vom Browser, weil Zertifikat nicht registriert

```
const fs = require('fs');
const key = fs.readFileSync('./certificates/key.pem');
const cert = fs.readFileSync('./certificates/cert.pem');

const express = require('express');
const https = require('https');

const app = express();

const server = https.createServer({key: key, cert: cert }, app);

app.get('/', (req, res) => { res.send('this is an secure server') });

server.listen(443, () => { console.log('listening on 443') });
```

Quelle: https://medium.com/@nitinpatel_20236/how-to-create-an-https-server-on-localhost-using-express-366435d61f28

Autorisierung und Integrität

Autorisierung

- Verfahren zur Sicherstellung, dass ein Nutzer nur jene Aktionen ausführen kann, zu denen er berechtigt ist
- In der Regel umgesetzt über Rechterollen, in denen spezifiziert ist, welche Transaktionen ein Nutzer mit der betreffenden Rechterolle ausführen darf
- Gängige Transaktionen
  - Anlegen, Speichern, Sehen und Löschen von Entitäten
  - Öffnen von Views/ Komponenten der Oberfläche
  - Ausführen von Aktionen (z.B. Abrechnen, Mails versenden, etc.)
- Häufig auch Vererbungshierarchien zwischen den Rechterollen
  - Transaktionsrechte einer Rolle können an andere Rollen „vererbt“ werden
- Nutzer können auch mehreren Rollen haben

Autorisierung

- Bei einfachen System reicht ggfls. nur eine Liste von Rollen zur Rechteprüfung
  - Wenig flexible
  - Langfristige Nutzung unwahrscheinlich
- Bei größeren System und Standardberechtigungskonzepten können existierende Module genutzt werden (z.B. accesscontrol)
- Bei größeren Systemen mit individuellen Rechteprofilen sind ggfls. Eigenimplementierungen notwendig

Autorisierung mit accesscontrol

- Rollbasierte Zugriffskontrolle
- Installation: **npm install accesscontrol**

```
const AccessControl = require('accesscontrol');

const ac = new AccessControl();
ac.grant('user')                    // define new or modify existing role. also takes an array.
   .createOwn('video')              // equivalent to .createOwn('video', ['*'])
   .deleteOwn('video')
   .readAny('video');

const permission = ac.can('user').createOwn('video');
console.log(permission.granted);    // —> true
console.log(permission.attributes); // —> ['*'] (all attributes)
```

Autorisierung

- Prüfung der Rechte

```
const ac = new AccessControl(grants);
// ...
router.get('/videos/:title', function (req, res, next) {
    const permission = ac.can(req.user.role).readAny('video');
    if (permission.granted) {
        Video.find(req.params.title, function (err, data) {
            if (err || !data) return res.status(404).end();
            // filter data by permission attributes and send.
            res.json(permission.filter(data));
        });
    } else {
        // resource is forbidden for this user/role
        res.status(403).end();
    }
});
```

Integrität

- Zur Sicherung der Integrität von Nachrichtenmuss ein Verfahren angewandt werden, mit dem festgestellt werden kann, ob eine Nachricht verändert wurde
- Klassischer Ansatz: **digitale Signatur**
  - In der Regel Nutzung von asymmetrischer Verschlüsselung
  - Es wird Hashwert der Nachricht berechnet und mit privaten Schlüssel signiert
  - Mit Verifikationsschlüssel (Public Key) wird Hashwert entschlüsselt
  - Es wird dann der Hashwert für die Nachricht selber berechnet und mit dem Hashwert aus der Signatur verglichen
  - Wenn Hashwerte gleich sind, ist die Nachricht nicht verändert worden, da die Signatur nur mit dem privaten Schlüssel erstellt werden kann.
  - Weiterer Effekt: der Sender kann nicht leugnen, die Nachricht verschickt zu haben (Non-Repudiation)
- Bei Tokens wird ähnlicher Ansatz verwendet, jedoch wird in der Regel eine symmetrische Verschlüsselung mit dem geheimen Schlüsse angewandt