UNIVERSITAS SCIENTIARUM SZEGEDIENSIS
UNIVERSITY OF SZEGED
Department of Software Engineering

Angular biztonság

Dr. Bilicki Vilmos
Szoftverfejlesztés Tanszék

Az előző előadás összefoglalója

- Angular HTTP
  - Szolgáltatás igénybevétele
  - JSON feldolgozása, ellenőrzése
  - Hiba kezelés
  - Hiba részletek
    - retry()
  - Observables és operátorok
  - Nem JSON adat kérése
  - POST, DELETE, PUT
  - A kérés konfigurálása - kérés, válasz elkapása
  - Előrehaldás események
  - XSRF
  - Saját süti, fejléc
- Service worker

A következő előadás témája

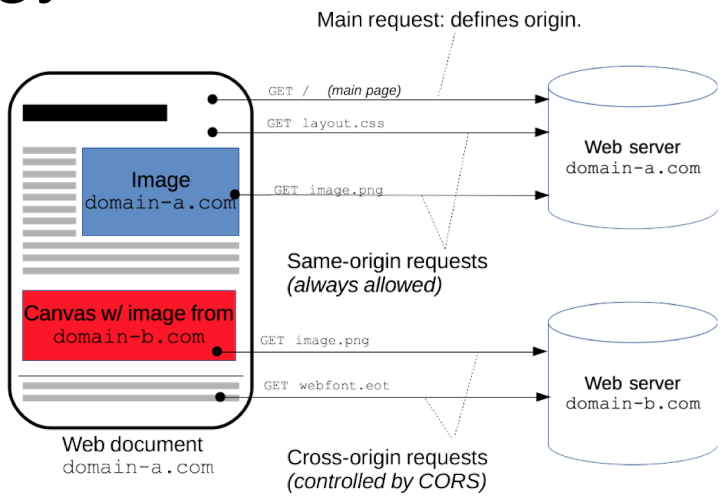
- Angular Biztonság
  - Legjobb gyakorlat
  - Cross-site scripting (XSS)
  - Biztonsági kontextus
  - DOM közvetlen használata és szeparáció
  - Tartalom biztonsági szabályok
  - Offline sablon fordító
  - Szerver oldali XSS védelem
  - Biztonságos értékek
  - HTTP sérülékenységek
  - Auditálás

Legjobb gyakorlat

- Igyekezzünk az Angular verziókat követni
  - Az Angular verziók (adott verzión belül is) biztonsági javításokat is tartalmaznak.
- Ne módosítsuk az Angular keretrendszert.
  - Saját, testreszabott Angular frissítése nehézkes,
  - Így a biztonsági javítások nyomonkövetése is problémás.
  - Célszerűbb a javításokat, módosításokat megosztani a közösséggel.
- A “Security Risk.”-ként megjelölt API-kat lehetőleg ne használjuk
  - Biztonságos értékek, lásd később

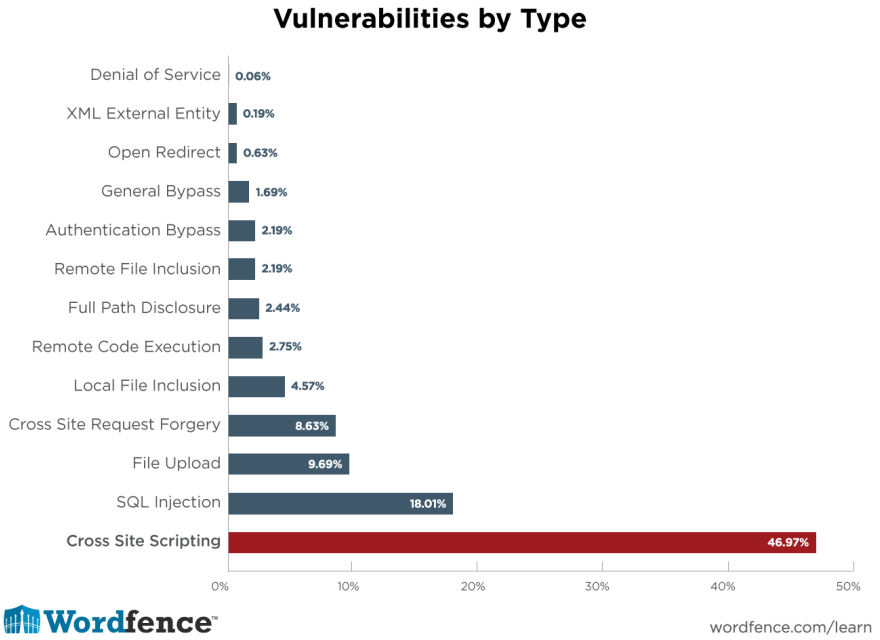
A Web biztonsági modellje

- Az azonos forrás szabályban testesül meg
  - A https://mybank.com –ból származó kódnak cask a https://mybank.com adatához férhet hozzá és a https://evil.example.com soha sem férhet hozzá a mybank adataihoz
  - Az egyes források egy-egy homokzóban vannak elkülönítve egymástól.



Top 5 biztonsági probléma

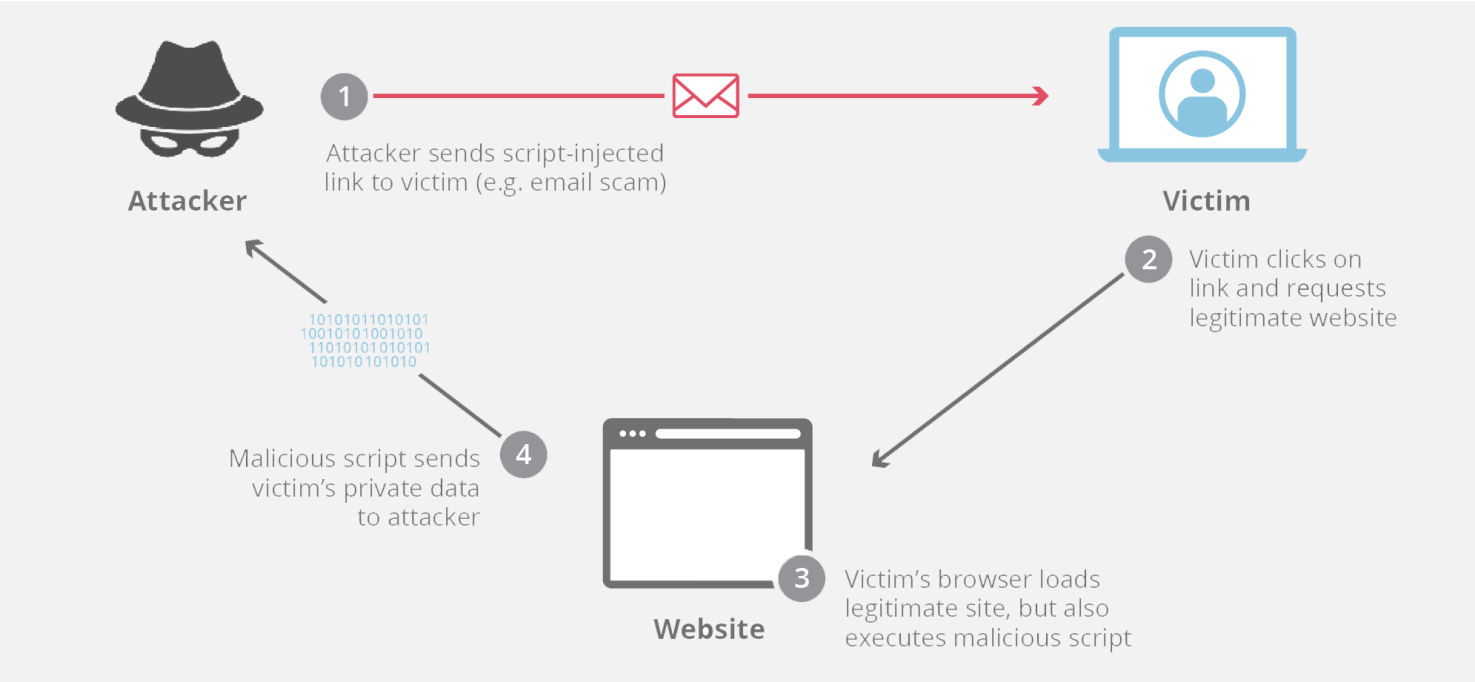
- Cross Site Scripting
- SQL Injection
- File Upload
- Cross Site Request Forgery
- Remote Code Execution

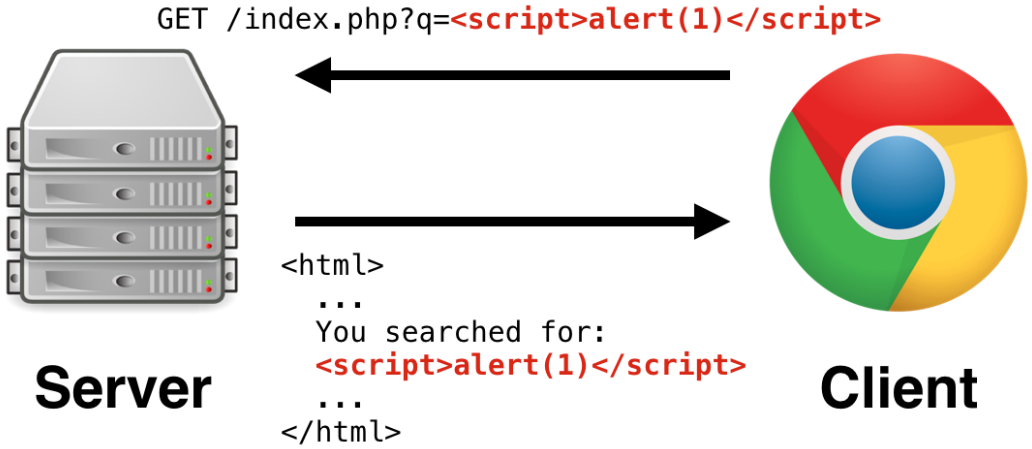


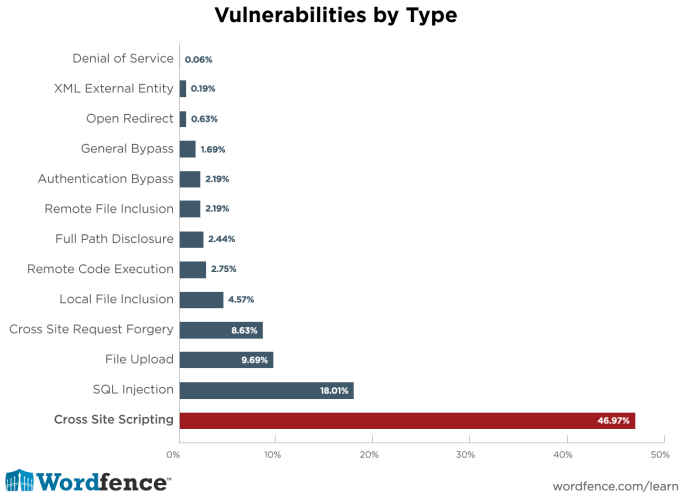
A cross-site scripting (XSS) megakadályozása

- A Cross-site scripting (XSS) lehetővé teszi a támadónak kártékony kódok beszúrását az adott weboldalba.
  - Az ilyen kód ellophatja a felhasználó adatait, vagy a felhasználó nevében cselekedhet.
  - A legnépszerűbb támadási forma
- Az XSS támadások kiévédésére a DOM-ot kell megvédenünk a nem megbízható kódoktól.
  - Ilyen lehet például, ha a támadónak sikerül egy <script> elemet becsempésznie a DOM-ba, ezzel tetszőleges kódot futtathat le.
  - Nem csak a <script> elem lehet problémás,
    - sok más elem vagy tuljadonság is tartalmazhat futtatandó kódot.
    - Ilyen lehet pl.: <img onerror="..."> és a <a href="javascript:...">.
  - Amennyiben a támadó adatot tehet be a DOM-ba akkor biztonsági kockázatnak vagyunk kitéve.

XSS


Attacker
1
Attacker sends script-injected link to victim (e.g. email scam)
Victim
2
Victim clicks on link and requests legitimate website
3
Victim's browser loads legitimate site, but also executes malicious script
Website
4
Malicious script sends victim's private data to attacker

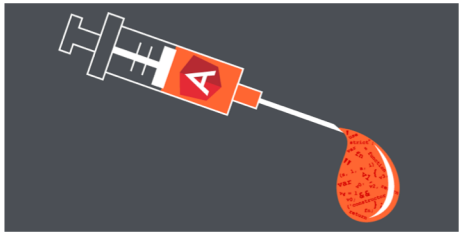

GET /index.php?q=<script>alert(1)</script>
<html>
...
You searched for:
<script>alert(1)</script>
...
</html>
Server
Client


Vulnerabilities by Type
Denial of Service 0.06%
XML External Entity 0.19%
Open Redirect 0.63%
General Bypass 1.69%
Authentication Bypass 2.19%
Remote File Inclusion 2.19%
Full Path Disclosure 2.44%
Remote Code Execution 2.75%
Local File Inclusion 4.57%
Cross Site Request Forgery 8.63%
File Upload 9.69%
SQL Injection 18.01%
Cross Site Scripting 46.97%
0%
10%
20%
30%
40%
50%
Wordfence
wordfence.com/learn

Az Angular cross-site scripting biztonsági modellje

- Az XSS biztonsági hiányosságok rendszerszintű megakadályozása érdekében az Angular alapértelmezés szerint nem megbízhatónak tart minden értéket.
  - Amikor tulajdonság, attribútum stílus, ... módon egy értéket szúrunk be a DOM-ba a sablonból akkor a kódot fertőtleníti és minden gyanús részt kihagy.
  - Az Angular sablonok is úgy kezeltek mint a kód de a HTML, attribútumok, kötő kifejezések megbízhatóak.
- Ne gyártsunk karaterlánc összefűzéssel sablont.
- Használjuk az offline sablon fordítót.

Fertőtlenítés és biztonsági kontextus



- A fertőtlenítés folyamata:
  - a nem megbízható adatokat/értékek vizsgálata és megbízható értékké konvertálása.
  - Ez már beszúrható a DOM-ba.
- A fertőtlenítés kontextusfüggő: egy a CSS-ben ártatlan érték veszélyes lehet az URL-ben
  - Fertőltelnítéskor
    - HTML alkalmaznak amikor az értéket HTML-be szúrhatják (pl.: innerHTML).
    - Stílust amikor a CCS-hez kötött értéket szúrunk be.
    - URL-t amikor URL adatot manipulálunk, pl.: <a href>.
    - Erőforrás URL egy olyan URL amely futtatható szkrpitre mutat rá <script src>.

Példa

- Az interpolált érték mindég lecserélődik – a HTML nincs interpretálva és a böngésző a az eleme szöveg kontextusában ábrázolja
- A HTML értelmezéséhez HTML tulajdonsághoz kell kötnünk pl.: innerHTML

```
<h3>Binding innerHTML</h3>
<p>Bound value:</p>
<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>
<p>Result of binding to innerHTML:</p>
<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>
```

Példa

```
<h3>Binding innerHTML</h3>
<p>Bound value:</p>
<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>
<p>Result of binding to innerHTML:</p>
<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>
```

```
export class InnerHtmlBindingComponent {
  // For example, a user/attacker-controlled value from a URL.
  htmlSnippet = 'Template <script>alert("Owned")</script> <b>Syntax</b>';
}
```


Result of binding to innerHTML:

Template **Syntax**

DOM API és explicit fertőtlenítés hívások

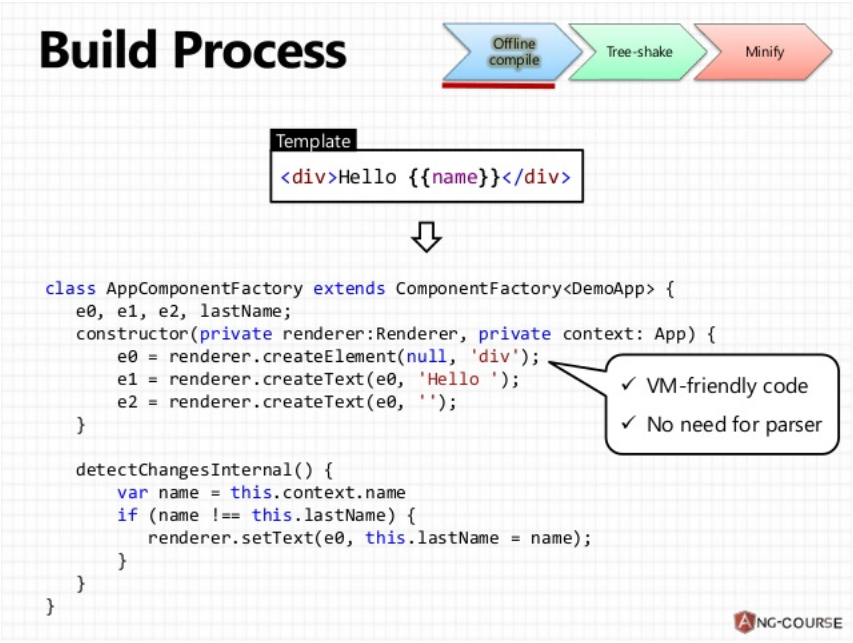
- A böngésző DOM API-ja nem véd meg feltétlenül a bizonstági támadásoktól
- Kerüljük a DOM közvetlen használatát, amennyiben lehet használjuk az Angular sablonokat
- Amikor nem tudjuk elkerülni a közvetlen DOM használatot akkor használjuk az Angular fertőtlenítő függvényeket.
  - Fertőtlenítsük a nem megbízható értékeket a DomSanitizer.sanitize metódussal és a megfelelő SecurityContext-tal.
- Itt a függvénynek megadhatjuk, hogy adott értékek megbízhatóak bypassSecurityTrust... ezek nem lesznek fertőtlenítve

Content Security Policy (CSP)

- Egy mély védekezési technológia az XSS kivédésére
- A CSP engedélyezéséhez, konfiguráljuk a böngészőt megfelelő Content-Security-Policy HTTP fejléc visszaadására
- A CSP kompatibilis böngésző csak akkor hajtja végre a szkriptet ha a forrás fájl engedélyezett doménről lett letöltve.
  - Minden más szkriptet mellőz (beleértve a HTML be ágyazott szkripteket is).
- A futtatható szkripteket adó doméneken túl a szerver a protokollokat is megaszbhatja;
  - például (és biztonsági szempontból ideális esteben), a szerver emegadhatja, hogy csak HTTPS-en lehet kiszolgálni
- Policy fájlok:
  - Content-Security-Policy: default-src 'self'
  - Content-Security-Policy: default-src 'self' *.trusted.com
  - Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com media2.com; script-src userscripts.example.com

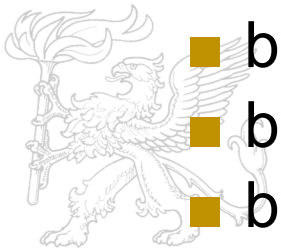
Az offline sablon fordító használata

- Az offline fordító használata egész támadás típusokat lehetlenít el (pl.: sablon injektálás). Az alkalmazás sebességére is jótékony hatással van
- Produkciós környezetben használjuk az offline sablon fordítót, ne alkalmazzunk dinamikus sablont
- Az Angular megbízik a sablon kódban, így ha felhasználói kódból generálunk sbalont akkor megkerüljük az Angular védelmét.



Megbízható értékek

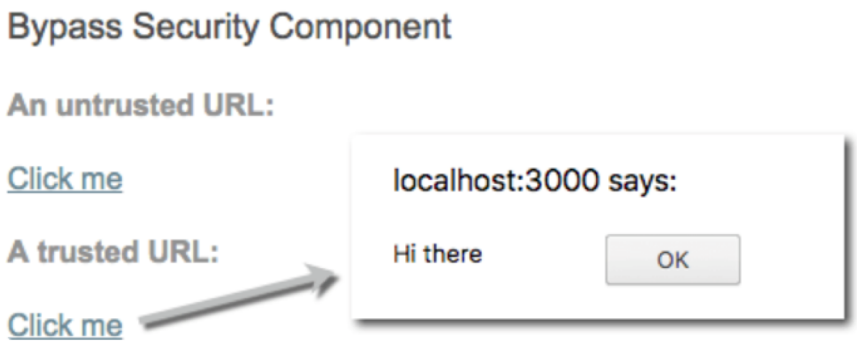
- Van olyan eset amikor futtatható kódot kell beszúrnunk (pl.: iframe)vagy veszélesnek tűnő URL-t kell összereknunk.
- Az automatikus fertőtlenítést ilyen esetekben meg tudjuk akadályozni
- Megbízhatónak kell ezen értkekekt jelölnünk. Ehhez injektáljuk a DomSanitizer-t és hívjuk meg adott metódusát:
  - bypassSecurityTrustHtml
  - bypassSecurityTrustScript
  - bypassSecurityTrustStyle
  - bypassSecurityTrustUrl
  - bypassSecurityTrustResourceUrl



Példa

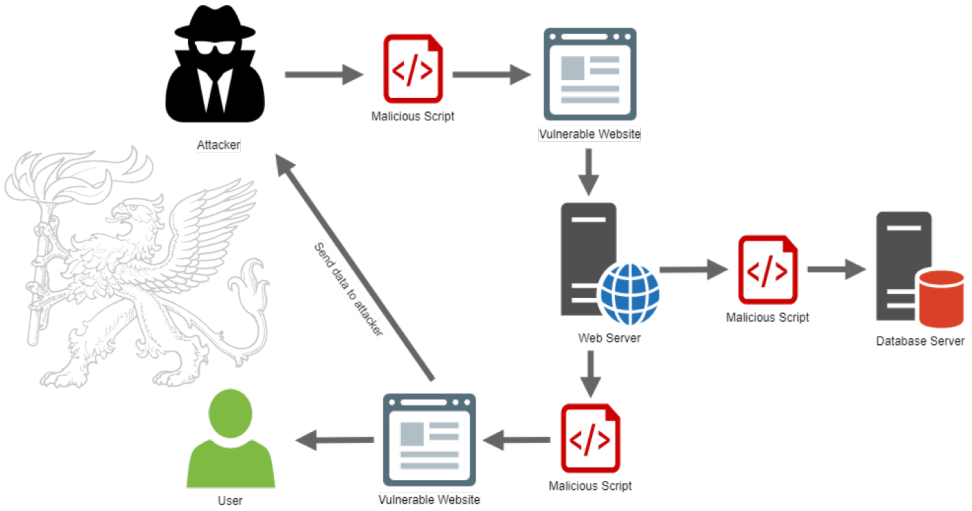

```
  <h4>An untrusted URL:</h4>
  <p><a class="e2e-dangerous-url" [href]="dangerousUrl">Click me</a></p>
  <h4>A trusted URL:</h4>
  <p><a class="e2e-trusted-url" [href]="trustedUrl">Click me</a></p>
```

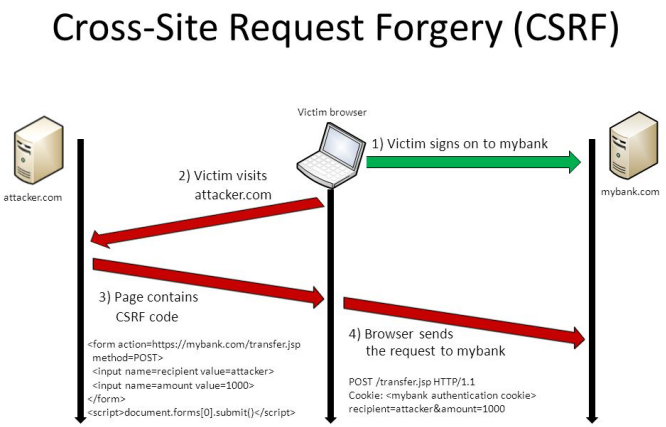
```
constructor(private sanitizer: DomSanitizer) {
  // javascript: URLs are dangerous if attacker controlled.
  // Angular sanitizes them in data binding, but you can
  // explicitly tell Angular to trust this value:
  this.dangerousUrl = 'javascript:alert("Hi there")';
  this.trustedUrl = sanitizer.bypassSecurityTrustUrl(this.dangerousUrl);
```

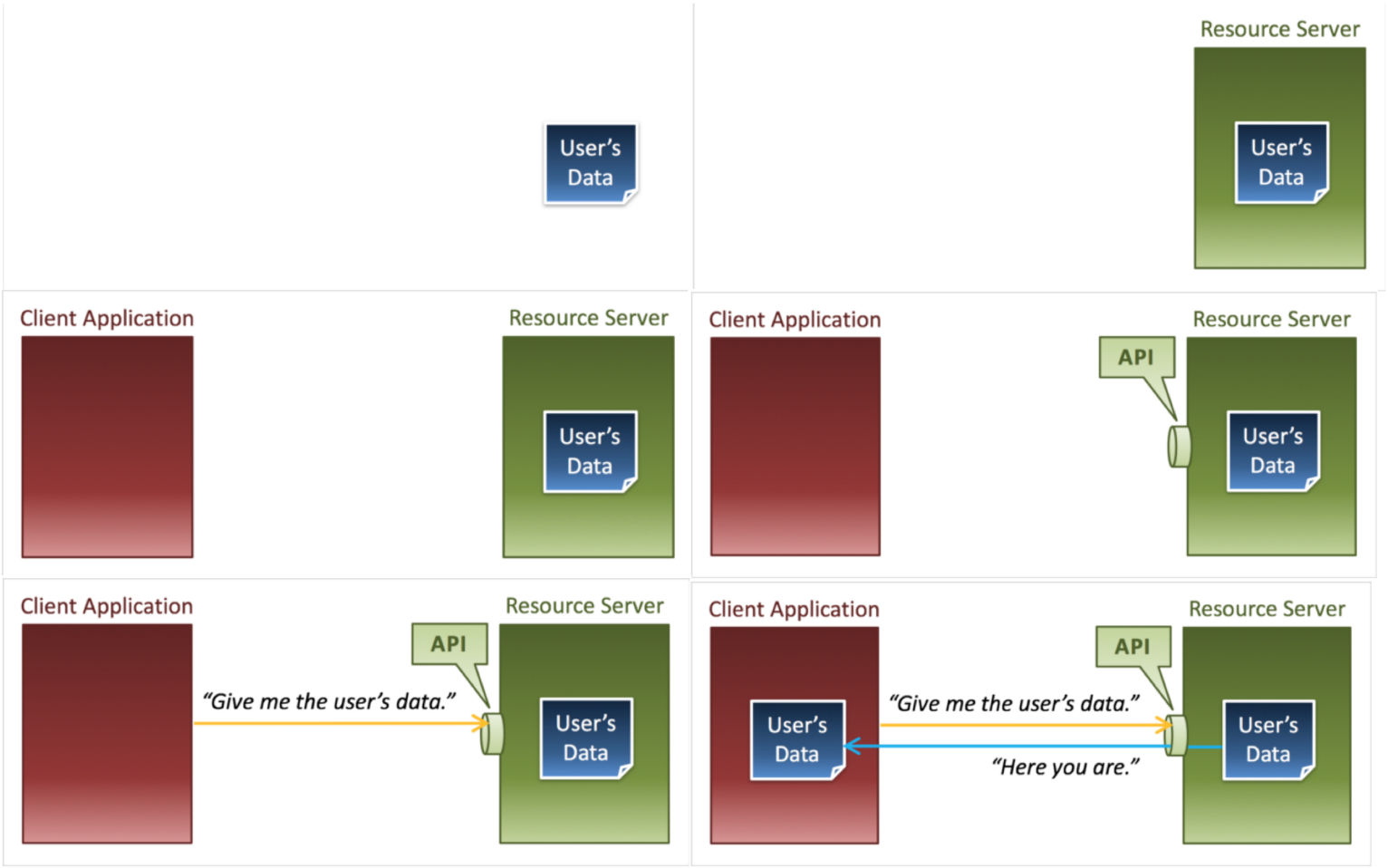
HTTP szintű sérülékenység

- Az Angular két HTTP szintű sérülékenységet tud a beépített megoldásaival kezelni
  - cross-site request forgery (CSRF or XSRF)
  - cross-site script inclusion (XSSI).





OAuth 2.0


User's Data
Resource Server
User's Data
Client Application
Resource Server
User's Data
Client Application
Resource Server
API
User's Data
Client Application
Resource Server
API
"Give me the user's data."
User's Data
Client Application
Resource Server
API
User's Data
"Give me the user's data."
"Here you are."
User's Data

OAuth 2.0


Malicious App
Resource Server
API
User's Data
"Give me the user's data."
"Here you are."
Even a malicious application can get the user's data.
Need a mechanism to protect the user's data!
Client Application
Access Token

OAuth 2.0

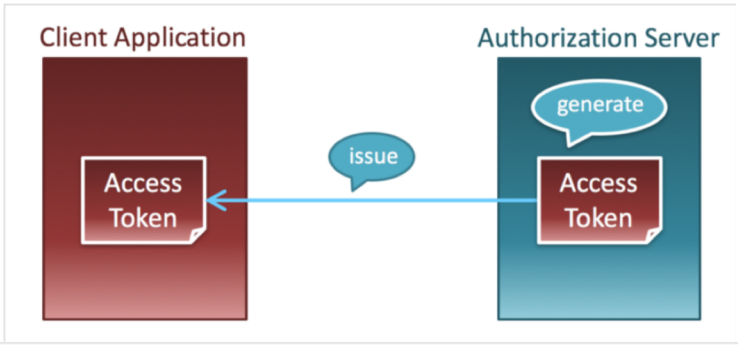

Client Application
Access Token
Access Token
"Give me the user's data."
API
Resource Server
User's Data
Client Application
Access Token
Access Token
"Give me the user's data."
API
Resource Server
Access Token
User's Data
Client Application
Access Token
Access Token
"Give me the user's data."
API
Resource Server
Access Token
verify
User's Data
Client Application
Access Token
User's Data
Access Token
"Give me the user's data."
"Here you are."
API
Resource Server
Access Token
verify
User's Data
Client Application
Access Token
User's Data
Access Token
"Give me the user's data."
"Here you are."
API
Resource Server
Access Token
verify
User's Data
An access token must be given to the client in advance.
Need someone who issues access tokens!
Client Application
Access Token
User's Data
Access Token
"Give me the user's data."
"Here you are."
API
Resource Server
Access Token
verify
User's Data
An access token must be given to the client in advance.

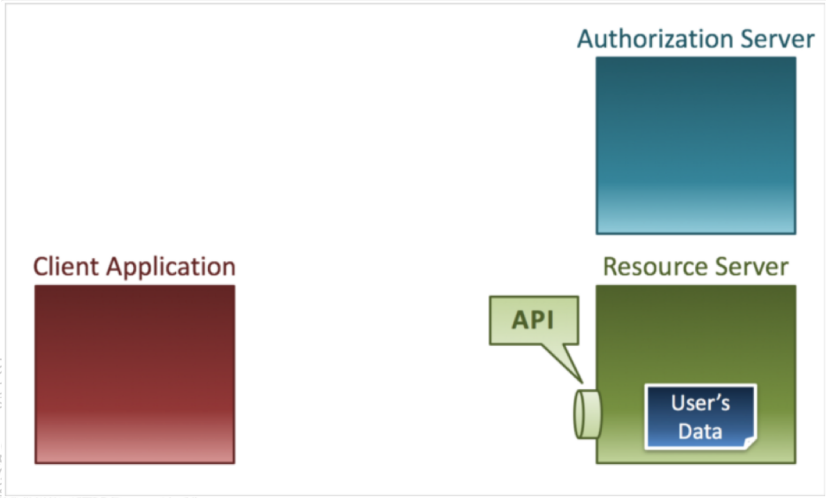
OAuth 2.0

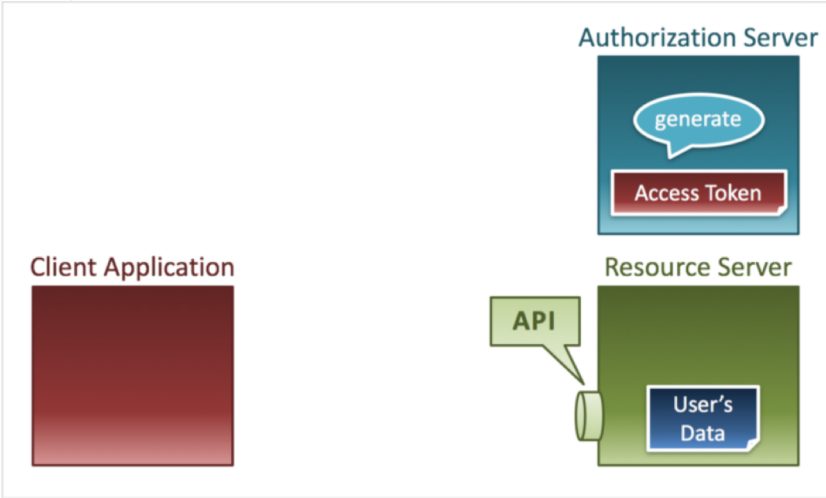
Someone who issues access tokens

||

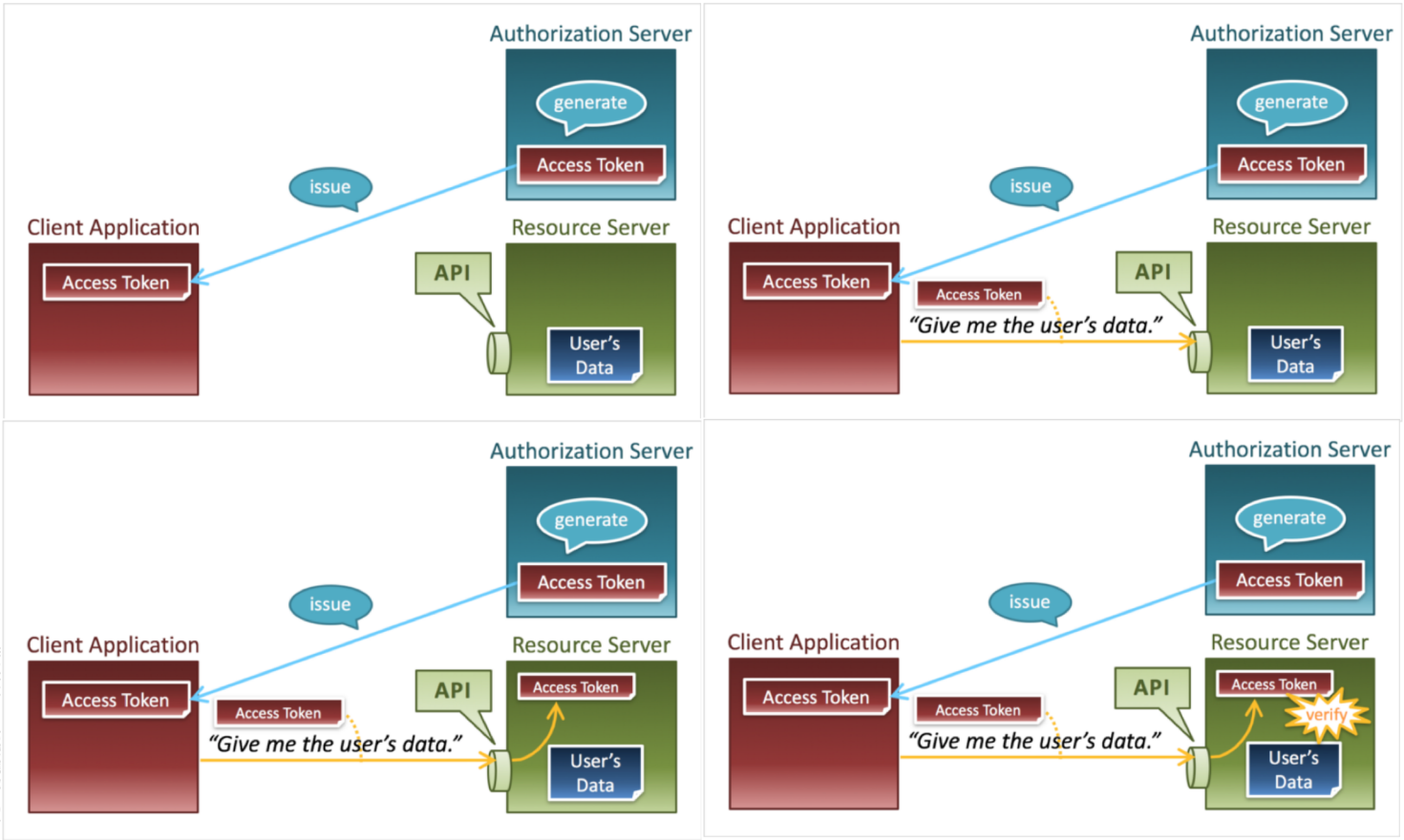
Authorization Server



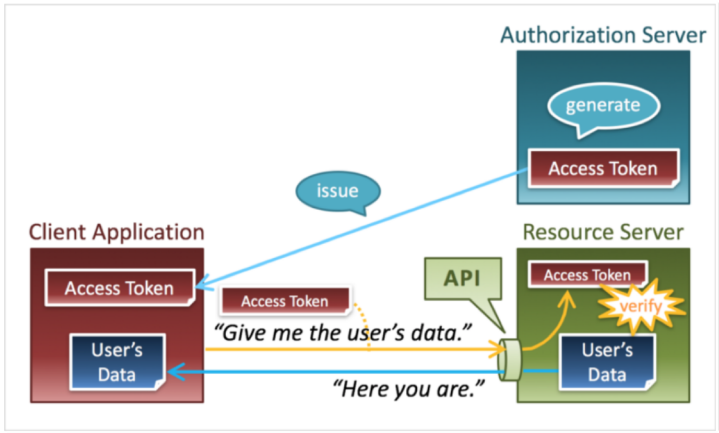


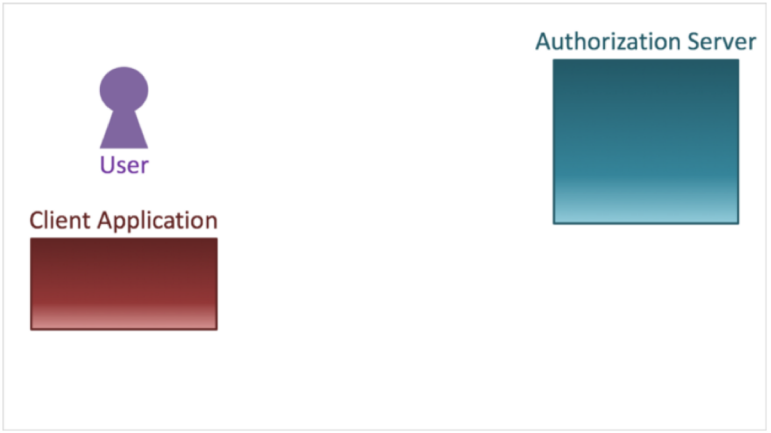


OAuth 2.0

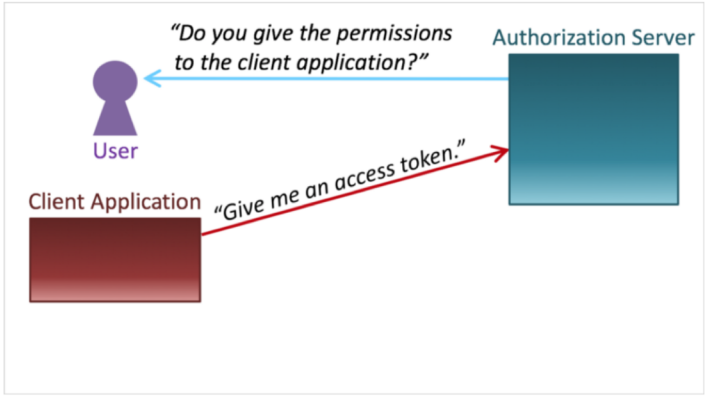

Authorization Server
generate
Access Token
issue
Client Application
Access Token
Resource Server
API
User's Data
Authorization Server
generate
Access Token
issue
Client Application
Access Token
Access Token
"Give me the user's data."
Resource Server
API
User's Data
Authorization Server
generate
Access Token
issue
Client Application
Access Token
Access Token
"Give me the user's data."
Resource Server
API
Access Token
User's Data
Authorization Server
generate
Access Token
issue
Client Application
Access Token
Access Token
"Give me the user's data."
Resource Server
API
Access Token
verify
User's Data

OAuth 2.0

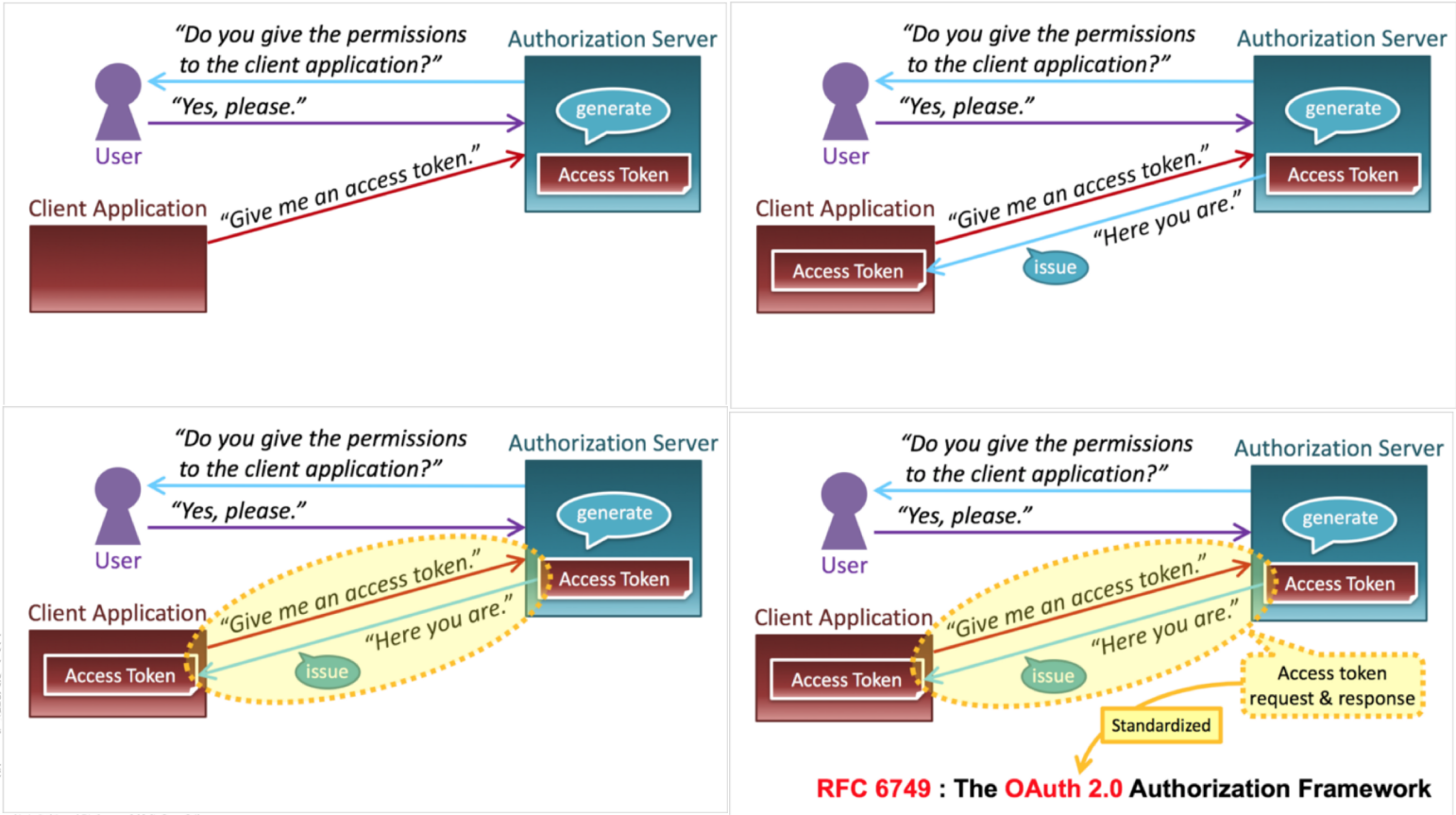

Authorization Server
generate
Access Token
issue
Client Application
Resource Server
Access Token
API
Access Token
verify
Access Token
"Give me the user's data."
User's Data
User's Data
"Here you are."


Authorization Server
User
Client Application

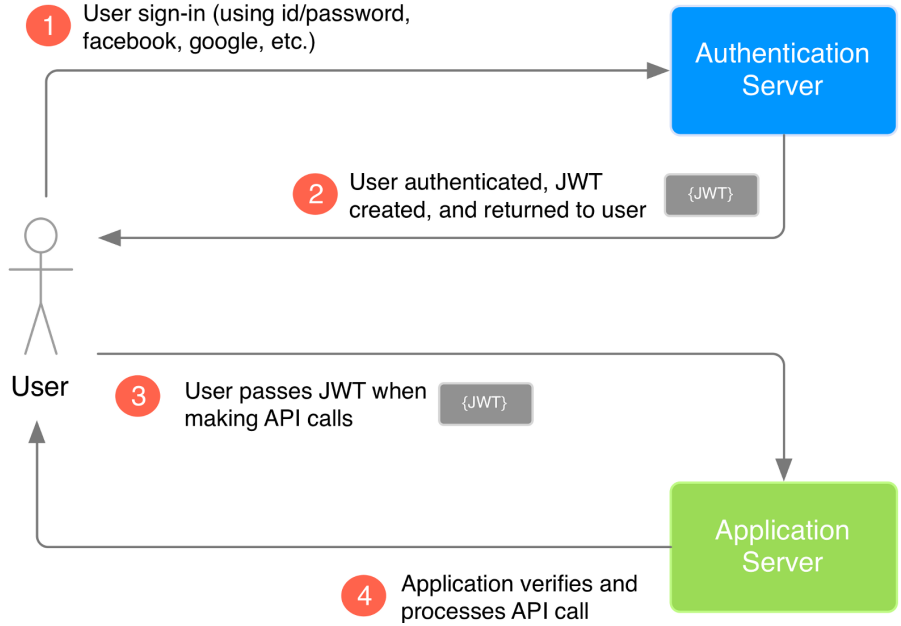

Authorization Server
User
Client Application
"Give me an access token."


"Do you give the permissions to the client application?"
Authorization Server
User
Client Application
"Give me an access token."

OAuth 2.0


"Do you give the permissions to the client application?"
"Yes, please."
User
Client Application
"Give me an access token."
Authorization Server
generate
Access Token
"Here you are."
issue
Access Token
Access token request & response
Standardized
RFC 6749 : The OAuth 2.0 Authorization Framework

JSON Web Token



A JSON Web Token (JWT) egy JSON objektum amelyet az RFC 7519 definiál. Segítségével biztonságos módon lehet adott értéket megadni két fél között. Az összetevői: fejléc, tartalom, aláírás.

Fontos megértenünk, hogy a JWT **NEM** információ elrejtésre van alkalmazva. A JWT-vel bizonytíani lehet, hogy az infomráció az és az adott forrástól származik. Az adat kódolva és nem titkosítva van.

Azonosítás a gyakorlatban

- 1. Lépés - Login oldal
- 2. Lépés – JWT alapú felhasználó viszony létrehozása
- 3. Lépés – A JWT visszaküldése a klinesnek
- 4. Lépés: A JWT tráolása
- 5. Lépés – A felhasználó kérés validálása

1. Lépés - A Login oldal

- A publikus Internet-en, a login oldal:
  - szolgáltatva lehet egy harmadik fél által pl.: Auth0
  - az alkalmazás része is lehet (SPA)

```
@Component({
  selector: 'login',
  template: `
<form [formGroup]="form">
    <fieldset>
        <legend>Login</legend>
        <div class="form-field">
            <label>Email:</label>
            <input name="email" formControlName="email">
        </div>
        <div class="form-field">
            <label>Password:</label>
            <input name="password" formControlName="password"
                   type="password">
        </div>
    </fieldset>
    <div class="form-buttons">
        <button class="button button-primary"
                (click)="login()">Login</button>
    </div>
</form>`})
```

```
export class LoginComponent {
    form:FormGroup;

    constructor(private fb:FormBuilder,
                private authService: AuthService,
                private router: Router) {

        this.form = this.fb.group({
            email: ['',Validators.required],
            password: ['',Validators.required]
        });
    }

    login() {
        const val = this.form.value;

        if (val.email && val.password) {
            this.authService.login(val.email, val.password)
                .subscribe(
                    () => {
                        console.log("User is logged in");
                        this.router.navigateByUrl('/');
                    }
                );
        }
    }
}
```

Azonosító szolgáltatás

- Érdemes az azonosító szolgáltatást egy kliens szintű singelton szolgáltatásba helyezni.
- Így később cserélehtő lehet az algoritmus, hely, ….

```
@Injectable()
export class AuthService {

    constructor(private http: HttpClient) {
    }

    login(email:string, password:string ) {
        return this.http.post<User>('/api/login', {email, password})
            // this is just the HTTP call,
            // we still need to handle the reception of the token
            .shareReplay();
    }
}
```

2. Lépés - JWT viszony jel létrehozása

- A cél a jelszó validálása és egy viszony létrehozása.
  - Amennyiben a jelszó megfelelő akkor a szerver létrehoz egy vivő jelet:
    - A jelhez a 35345435435435345 3 technikai ID-jű felhasználó tartozik és a viszony két óráig érvényes


```
const RSA_PRIVATE_KEY = fs.readFileSync('./demos/private.key');

export function loginRoute(req: Request, res: Response) {

    const email = req.body.email,
          password = req.body.password;

    if (validateEmailAndPassword()) {
       const userId = findUserIdForEmail(email);

        const jwtBearerToken = jwt.sign({}, RSA_PRIVATE_KEY, {
                algorithm: 'RS256',
                expiresIn: 120,
                subject: userId
            }

          // send the JWT back to the user
          // TODO - multiple options available
    }
    else {
        // send status 401 Unauthorized
        res.sendStatus(401);
    }
}
```

3. Lépés JWT elküldése a kliensnek

- Több lehetséges megoldás között választhatunk:
  - Sütiben
  - A kérés törzsében
  - Egyszerű HTTP fejlécben

JWT-k és Sütik

- A sütik egyedi képessége az, hogy a böngésző minden kéréshez automatikusan hozzácsapja (adott tartományban)
- Ez alapján amennyiben sütiben tároljuk a JWT-t akkor nem kell kliens logika az azonosítás elküldéséhez


```
... continuing the implementation of the Express login route

// this is the session token we created above
const jwtBearerToken = jwt.sign(...);

// set it in an HTTP Only + Secure Cookie
res.cookie("SESSIONID", jwtBearerToken, {httpOnly:true, secure:true});
```

A sütik biztonsági tuljadonságai HttpOnly és Secure Flags

- A sütik biztonsági szinteket definiáló tulajdonságokkal bírnak.
- Secure: ez esetben a sütit csak akkor küldi el a böngésző ha HTTPS kapcsolaton van.
- Http Only: esteén a süti nem érehető el JS-ből (a böngésző továbbra is küldi).
  - Ezzel kezelhető a szkript injektálásos támadás vagy XSS. A támadó nem fér hozzá a sütihez (nem tud a felhasználó nevében beszélni)

A sütik hátrányai - XSRF

- Amennyiben sütiben tároljuk a JWT-t akkor a Cross-Site Request Forgery, vagy XSRF / CSRFtámadással szemben védtelen:
  - valaki küld nekünk egy linket amire rákattintunk
  - A link egy HTTP kéréssel végződik a támadott weboldalhoz (ide ugye megvan minden sütink)
  - Amennyire ide már be vagyunk jelentkezve akkor a JWT- tartalmazó sütit is elküldi
  - A szevrer egy érvényes JWT-t kap, így nem tudja megkülönböztetni, hogy ez egy jogos vagy kártékony kérés volt

A HTTP válasz törzs használata

- Nem csak A JWT-t de a lejárati időt is elküldhetjük


```
... continuing the implementation of the Express login route

// this is the session token we created above
const jwtBearerToken = jwt.sign(...);

// set it in the HTTP Response body
res.status(200).json({
  idToken: jwtBearerToken,
  expiresIn: ...
});
```

4. Lépés – A JWT tárolása és felhasználása

- Egy praktikus hely a Local Storage, amely egy név/érték tár.

```
import * as moment from "moment";

@Injectable()
export class AuthService {

    constructor(private http: HttpClient) {

    }

    login(email:string, password:string ) {
        return this.http.post<User>('/api/login', {email, password})
            .do(res => this.setSession)
            .shareReplay();
    }

    private setSession(authResult) {
        const expiresAt = moment().add(authResult.expiresIn,'second');

        localStorage.setItem('id_token', authResult.idToken);
        localStorage.setItem("expires_at", JSON.stringify(expiresAt.valueOf()) );
    }

    logout() {
        localStorage.removeItem("id_token");
        localStorage.removeItem("expires_at");
    }

    public isLoggedIn() {
        return moment().isBefore(this.getExpiration());
    }

    isLoggedOut() {
        return !this.isLoggedIn();
    }

    getExpiration() {
        const expiration = localStorage.getItem("expires_at");
        const expiresAt = JSON.parse(expiration);
        return moment(expiresAt);
    }
}
```

A JWT hozzácsapása a kérésekhez

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

    intercept(req: HttpRequest<any>,
              next: HttpHandler): Observable<HttpEvent<any>> {

        const idToken = localStorage.getItem("id_token");

        if (idToken) {
            const cloned = req.clone({
                headers: req.headers.set("Authorization",
                    "Bearer " + idToken)
            });

            return next.handle(cloned);
        }
        else {
            return next.handle(req);
        }
    }
}
```

JWKS (JSON Web Key Set) végpontok és kulcs rotálás

- JWKS vagy JSON Web Key Set egy JSON alapú szabvány a publikus kulcsok REST alapú publikálására endpoint.

```
{
  "keys": [
    {
      "alg": "RS256",
      "kty": "RSA",
      "use": "sig",
      "x5c": [
        "MIIDJTCCAg2gAwIBAgIJUP6A\/iwWqvedMA0GCSqGSIb3DQEBCwUAMDAxLjAsBgNVBAMTJWFuZ3VsYXJ1b
      ],
      "n": "wUvZ-4dkT2nTfCDIwyH9K0tH4qYMGcW_KDYeh-TjBdASUS9cd741C0XMvmVSYGRP0BOLeXeaQaSdKBi
      "e": "AQAB",
      "kid": "QzY0NjREMjkyQTI4RTU2RkE4MUJBRDExNzY1MUY1N0I4QjFCODlBOQ",
      "x5t": "QzY0NjREMjkyQTI4RTU2RkE4MUJBRDExNzY1MUY1N0I4QjFCODlBOQ"
    }
  ]
}
```

```
const jwksRsa = require('jwks-rsa');
const expressJwt = require('express-jwt');

const checkIfAuthenticated = expressJwt({
    secret: jwksRsa.expressJwtSecret({
        cache: true,
        rateLimit: true,
        jwksUri: "https://angularuniv-security-course.auth0.com/.well-known/jwks.json"
    }),
    algorithms: ['RS256']
});

app.route('/api/lessons')
    .get(checkIfAuthenticated, readAllLessons);
```

Összefoglaló

- Angular Biztonság
  - Legjobb gyakorlat
  - Cross-site scripting (XSS)
  - Biztonsági kontextus
  - DOM közvetlen használata és szeparáció
  - Tartalom biztonsági szabályok
  - Offline sablon fordító
  - Szerver oldali XSS védelem
  - Biztonságos értékek
  - HTTP sérülékenységek
  - Auditálás

A következő előadás tartalma

- Angular Tesztelés
  - Set up continuous integration
  - Configure project for Circle CI
  - Configure project for Travis CI
  - Configure CLI for CI testing in Chrome
  - Enable code coverage reports
  - Code coverage enforcement
  - Service Tests
  - Component Test Basics
  - Component class testing
  - Component DOM testing
  - Component Test Scenarios
  - Component binding
  - Component with external files
  - Component with a dependency
  - Component with async service
  - Component marble tests
  - Component with inputs and outputs
  - Component inside a test host
  - Routing component
  - Routed components
  - Nested component tests
  - Components with RouterLink
  - Use a page object
  - Calling compileComponents()
  - Setup with module imports
  - Override component providers
  - Attribute Directive Testing
  - Pipe Testing
  - Test debugging
  - Testing Utility APIs