Sicherheit in SPAs

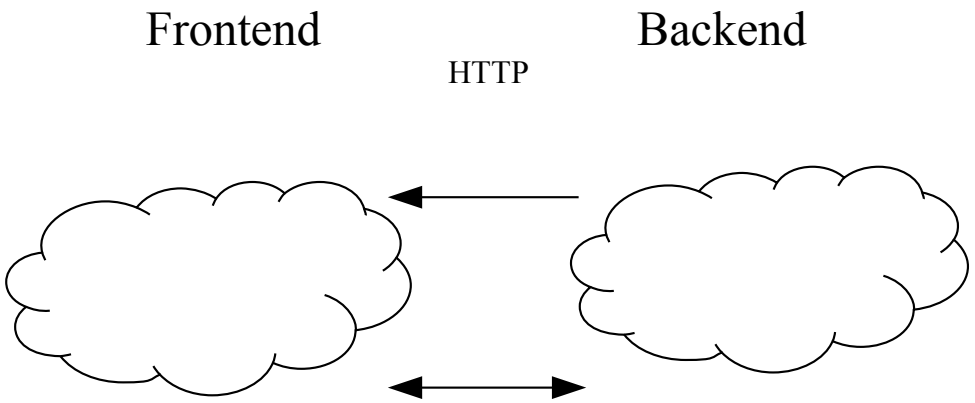
mit

- Ausgangssituation
- Sicherheitskonzept
- Gängige Probleme
  - Beispiel
  - Gegenmaßnahmen

ÜBER MICH

- Philipp Burgmer
  - Software-Entwickler, Trainer
  - Fokus: Frontend, Web-Technologien
  - burgmer@w11k.de
- w11k GmbH
  - Software Design, Entwicklung & Wartung
  - Consulting, Schulungen & Projekt Kickoff
  - Web-Apps, Mobil-Apps, Rich Clients
  - AngularJS, TypeScript, Eclipse RCP

ARCHITEKTUR VON SPAs



- Rich Client im Browser
- Backend weis nichts über verwendete Technologien im Client
- Client weis nichts über verwendete Technologien im Backend
- Stateful Client, Stateless Backend

TECHNOLOGIEN

- Backend: völlig frei (SQL, NoSQL, Java, .NET, …)
- Kommunikation: HTTP
- Frontend: Browser mit HTML, JavaScript & CSS

- Historisch betrachten
- Vieles gewachsen
- Nicht für heute Verwendung gedacht

SICHERHEITSKONZEPT

NAIV

- Grundgedanke: Zwei getrennte Systeme
- Jeder sichert sich selbst ab
  - Client schütz UI
  - Server schütz Datenzugriffe
  - Jeder schützt seine verwendeten Technologien
- Alle schützen die Übertragung

GENERELLE GEGENMASSNAHMEN

- Benutzereingaben nie trauen
  - Kommt Request vom Client? Mit validen Daten?
  - Was gibt der Anwender für Daten ins Formular ein?
  - Welche URL ruft der Anwender von Hand auf?
- Security testen
  - Grundlegend: Entwicklern selbst / aus anderem Team
  - Tiefgründig: Spezialisten

GENERELLE GEGENMASSNAHMEN

HTTP HEADER

- Browser-Verhalten kann per HTTP Header konfiguriert werden
- Nicht auf Defaults verlassen (ähnlich wie Reset-CSS)
- List of Useful HTTP Headers
  - `Strict-Transport-Security: max-age=86400; includeSubDomains`
  - `X-Frame-Options: deny`
  - `Content-Security-Policy: default-src 'self'`
  - `X-XSS-Protection: 1; mode=block`
  - `X-Content-Type-Options: nosniff`

TOP 10 SICHERHEITSPROBLEME

1. Injection
2. Broken Authentication and Session Management
3. Cross-Site Scripting
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

Quelle: OWASP Top10 2013

OWASP

- The Open Web Application Security Project
- Stellt Informationen zu Sicherheitsthemen bereit
  - detaillierte Beschreibungen und Erklärungen
  - gängige Lösungsansätze
- Non-Profit Organisation
- Finanziert über Mitgliedsbeiträge und Spenden
- Existiert seit 2001

A1

INJECTION

BEISPIEL

SQL

Java Code um SQL Abfrage zusammen zu bauen

```
statement = "SELECT * FROM users WHERE id = " + request.getParameter("id") + ";"
```

URL-Aufruf des Angreifers

```
http://example.com/user?id=42;UPDATE+USER+SET+TYPE="admin"+WHERE+ID=23;--
```

Ausgeführtes SQL

```
SELECT * FROM users WHERE id = 42; UPDATE USER SET TYPE="admin" WHERE ID=23;--;
```

BEISPIEL

HTML & DOM

```
1 var source = $('#insecure-input');
2 var text = source.val();
3 var target = $('#insecure-output');
4 target.append(text);
```

Ausprobieren ...

INJECTION

- Daten aus Sprache A werden zu Code in Sprache B
- Code wird dynamisch an einen Interpreter übergeben
- Code enthält Benutzereingaben (Formular-Daten, URL-Parameter, …)
- Benutzereingaben werden nicht oder unzureichend überprüft
- An vielen Stellen möglich
  - SQL
  - HTML (Content-Spoofing und Cross-Site-Scripting)
  - Script-Sprachen mit eval-Funktion (JS, PHP)
  - Dynamisches Laden von Code aus Dateien
  - Shell / Command Execution

GEGENMASSNAHMEN

- Möglichst wenig Interpreter verwenden
- APIs ohne Interpreter verwenden (z.B. Prepared-Statement)
- Benutzereingaben nicht vertrauen
  - Kontextuelles Escapen (HTML, JS, SQL)
  - White-Listing

GEGENMASSNAHME

SQL

Sicherer Java Code um SQL Abfrage zusammen zu bauen

```
1 PreparedStatement pstmt = connection.prepareStatement("SELECT * FROM users WHERE id = ?");
2 pstmt.setInt(1, request.getParameter("id"));
3 ResultSet rset = pstmt.executeQuery();
```

GEGENMASSNAHME

HTML & DOM

```
1  var source = $('#insecure-input');
2  var target = $('#insecure-output');
3
4  var newChild = $('<p></p>');
5  target.append(newChild);
6
7  var text = source.val();
8  newChild.text(text);
```

Ausprobieren ...

ANGULARJS

- Angular escapt alle Data-Bindings automatisch
- $sanitize Service um sicheres HTML-Subset ausgeben zu können
- $sce Service um beliebiges HTML aus vertrauenswürdiger Quelle ausgeben zu können
- Ausführliches Beispiel

ANGULARJS

BESPIEL

```
1 <input type="text" ng-model="text"/>
2 <div ng-bind="text"></div>
3 <div ng-bind-html="text"></div>
```

ANGULARJS

NG-BIND-HTML

- `ng-bind` und `{{}}` escaped alle HTML Sonderzeichen
- `ng-bind-html` lässt ein sicheres Subset durch

- `ngSanitize`: zusätzliches Modul mit erweitertem Sanitizer für sicheres Subset
- Muss eigebunden werden für `ng-bind-html`, ansonsten Fehler auf Konsole

ANGULARJS

STRICT CONTEXTUAL ESCAPING

- `$sce` Service stellt Methoden zum wrappen bereit
- JS, URL, HTML

- `$sce.trustAsHtml` wrapt Text in Objekt
- Objekt markiert Text als sicheren Code
- `ng-bind-html` übernimmt ursprünglichen Text als Code in DOM

ANGULARJS

`$interpolate`

> *For security purposes, it is strongly encouraged that web servers escape user-supplied data, replacing angle brackets (<, >) with < and > respectively, and replacing all interpolation start/end markers with their escaped counterparts.*
>
> **$interpolate API Documentation at angularjs.org**

```
1 <input type="text" ng-model="text_1"/><span ng-bind="text_1"></span>
2 <input type="text" ng-model="text_2"/><span ng-bind="text_2"></span>
```

ANGULARJS

`$interpolate`

```
1 <div ng-init="text_1 = {{ text_2 = 1 }}">
2   <input type="text" ng-model="text_1"/><span ng-bind="text_1"></span>
3 </div>
4 <div>
5   <input type="text" ng-model="text_2"/><span ng-bind="text_2"></span>
6 </div>
```

- Daten vom Server, die durch `$interpolate` laufen, müssen escaped werden
- Soll der Server wirklich etwas über die Verwendung im Frontend wissen?
- Kann auch im Client per HTTP-Interceptor gemacht werden

A2

BROKEN AUTHENTICATION AND SESSION MANAGEMENT

BEISPIELE

- Passwörter stehen im Klartext in der Datenbank
  - Datenbank wird entwendet
  - Angreifer kann sich als jeder User einloggen
- Session-IDs kommen aus Pool
- Session ID wird entwendet (z.B. per XSS)

KLASSISCHES

SESSION MANAGEMENT

- Ausgangssituation
  - Anwender loggt sich ein
  - Server vergibt Session ID, überträgt als Cookie
  - Browser sendet Cookie mit Session ID automatisch mit
  - Server mappt Session ID auf User
- Probleme
  - Session kann geklaut werden (ID in URL, XSS)
  - Kein Session-Timeout (öffentlicher PC)
  - Vorhersagbare Session IDs
  - Übertragung per unverschlüsselter Kommunikation (Man-in-the-Middle)

GEGENMASSNAHMEN

SESSION MANAGEMENT

- Session Managemnt nicht selbst implementieren
- Bewährte, gut getestete Biblotheken verwenden
- Verschlüsselte Kommunikation (für alles!)
- Keine Passwörter im Klartext speichern, sondern Hash mit Salt
- Session-Cookie mit `httpOnly` und `secure`

SESSION MANAGEMENT

PROBLEME

- Zustand im Backend -> schlechte Skalierbarkeit
- Backend stellt API für verschiedene / mehrere Clients
- Backend stellt API für externen Dienst
  - Externen Dienst soll Passwort nicht kennen
  - Anwender soll Dienst authorisieren können
- Cookies werden nicht an andere Domainen geschickt (CDN)

TOKEN BASIERTE AUTHENTIFIZIERUNG

- Keine Session ID mit Mapping auf User im Server
- Server vergibt nach Login Token
- Token enthält alles was Server für Auth-Prüfung braucht
  - User-ID
  - Login-Timestamp
  - Expires
  - Optional: Rollen
- Token ist gegen Manipulation geschützt (JSON Web Token)
  - Verschlüsselung oder Message Authentication Code (z.B. HMAC)
  - Secret nur auf Server bekannt
- Client sendet Token bei jedem Request (Cookie oder Header)

TOKEN BASIERTE AUTHENTIFIZIERUNG

- +Stateless Backend
- +Cross-Domain-API-Calls
- +Anbindung externe Dienste (wenn Token Client gebunden)

- –Kein echter Logout möglich, nur Daten löschen im Client
- –Anfällig für Replay-Attacken

COOKIES vs HTTP HEADER

- Unabhängig von *Session vs Token*
- Token kann auch per Cookie gesendet werden

COOKIE vs HTTP HEADER

COOKIE

- +Werden automatisch übertragen
- –Aber nicht zu anderer Domain (CDN)
- –Immer, auch wenn nicht gewollt (XSRF)
- +Überlebt Reload der Anwendung
- +Kann vor Zugriff per JS geschützt werden

COOKIES vs HTTP HEADER

HTTP HEADER

- Kann von App in jedem Request gesetzt werden
- +Auch für Cross-Domain-Requests
- +Wird nicht automatisch gesendet (kein XSRF)
- +Mobile Ready (Native Apps, schlechter Support für Cookies)
- –Token muss persistiert werden (localStorage, pro Domain)
- –Anfällig für XSS

GEGENMASSNAHMEN

TOKEN BASIERTE AUTHENTIFIZIERUNG

- Token Authentifizierung nicht selbst implementieren
- Bewährte, gut getestete Biblotheken verwenden

- OAuth2 mit OpenID Connect
  - Implementierungen für Java und JavaScript vorhanden
  - AngularJS Module für Integration vorhanden
  - –Aufbau und Ablauf nicht trivial
  - +Wenn es mal läuft: sehr sicher

LOGIN

- Login vor Aufruf der Anwendung
- Login innerhalb der Anwendung

LOGIN

VOR DER ANWENDUNG

- Server stellt sicher
  - Anwendung nur mit gültigem Login aufrufbar
  - Ohne gültigen Login -> HTTP-Redirect auf Login-Seite
  - Nach erfolgreichem Login -> HTTP-Redirect auf Anwendung
- In Anwendung
  - Prüfen auf HTTP 401 -> Navigation zu Login-Seite

- +Weniger Angriffsfläche: Nicht jeder sieht die Anwendung
- +Schnelles Laden der ersten Seite
- –Immer ganze Anwendung geschützt

LOGIN

IN DER ANWENDUNG

- Rein Client-seitiges Handling (für UI)
- Login-Formular als Route / State in Anwendung
- Ajax-Request für Login
- Prüfung auf gültigen Login
  - State-Change + Event-Handler `$stateChangeError`
  - API-Requests + HTTP Interceptor

- +Weniger Request notwendig
- +Öffentliche und geschützte Bereiche möglich

A3

XSS

CROSS-SITE-SCRIPTING

BEISPIEL

```
1 var source = $('#insecure-input');
2 var text = source.val();
3 var target = $('#insecure-output');
4 target.append(text);
```

Ausprobieren ...

CROSS-SITE-SCRIPTING

- Spezielle Art der HTML Injection
- HTML-Injection wird ausgenutzt um anderen Benutzer Code unterzuschieben
- Benutzereingabe wird ohne Prüfung in HTML ausgegeben
- Ermöglicht Ausführen von Code

- Angriffe
  - Daten auslesen und an Angreifen übermitteln (z.B. Session-Cookie)
  - Code ruft URL auf um Aktion mit Rechten des Benutzers auszuführen (ähnlich wie XSRF)

GEGENMASSNAHMEN

- Wie bei *A1 - Injection*
  - Benutzereingaben immer escapen
  - Daten vom Server escapen
  - Sanitizer Biblothek verwenden
  - Kontext beachten in dem Wert verwendet wird
- Content-Security-Policy anwenden

CONTENT SECURITY POLICY

- What is CSP and Why Haven't You Applied it Yet
- Per Header einschalten: `Content-Security-Policy: default-src 'self'`
- Verhindert Ausführen von *Text als Code* (JavaScript `eval`)
- Kein dynamisches Erzeugen von Script- und Style-Tags

CONTENT SECURITY POLICY

ANGULARJS

- Attribut bei ngApp

```
1 <html ng-app="app" ng-csp></html>
```

- AngularJS verwendet kann kein `new Function` mehr
- Langsamer, aber sicherer
- AngularJS kann kein CSS mehr in Seite injizieren
  -> `angular-csp.css` einbinden

A7

MISSING FUNCTION LEVEL ACCESS CONTROL

BEISPIEL

- Benutzer ist kein Admin
- Admin-Bereich wird im Menü nicht angezeigt
- Admin-Bereich ist aber über URL erreichbar

```
http://example.com/app/admin
```

- Angreifer kann API direkt aufrufen

```
http://example.com/api/admin/users
```

MISSING FUNCTION LEVEL ACCESS CONTROL

- Was kann der Anwender wann wo aufrufen?
- Wird dabei seine Berechtigung geprüft?

GEGENMASSNAHMEN

- Berechtigungen über Rollen/Gruppen verwalten
- Berechtigungen an allen relevanten Stellen prüfen
- Im Client
  - Im UI nicht anzeigen was der User eh nicht Ausführen darf
  - Manuelle Ausführung verhindern
- Im Server
  - Requests vom Client nicht vertrauen
  - An jedem REST Endpoint Berechtigungen prüfen
  - Eventuell pro Verb (jeder darf Lesen, nur Admin darf Schreiben)

BERECHTIGUNGEN

ANGULARJS

- Bereiche im Frontend mit Rollen versehen
- Im UI per Direktive

```
<ul class="menu">
  <li user-role-required="'ADMIN'">
    <a href="#!/admin">Admin</a>
  </li>
</ul>
```

- Bereich zusätzlich vor manuellem Aufruf schützen

BERECHTIGUNGEN

ANGULARJS

```
module.config(function() {
  $stateProvider.state('admin', {
    url: '/admin',
    templateUrl: 'route/admin/admin.html',
    data: {
      userRoleRequired: 'ADMIN'
    }
  });
});
```

BERECHTIGUNGEN

ANGULARJS

```
module.run(function($rootScope, UserService, growl) {
  $rootScope.$on('$stateChangeStart', function (event, toState) {
    if (toState.userRoleRequired) {
      UserService.hasRoles(toState.userRoleRequired).then(function () {
        // to late to cancel event
        event.preventDefault();
      });
    }
  });
});
```

BERECHTIGUNGEN

ANGULARJS

- An Route / State per `resolve`
- Event-Handler für `$stateChangeError`

```
angular.module('app').config(function() {
  $stateProvider.state('admin', {
    url: '/admin',
    templateUrl: 'route/admin/admin.html',
    resolve: {
      authorized: /* @ngInject */ function (UserService) {
        return UserService.hasRoles('ADMIN');
      }
    }
  });
});
```

A8

XSRF

CROSS-SITE-REQUEST-FORGERY

BEISPIEL

- Ausgangsituation: Benutzer in App eingeloggt (hat gültiges Session-Cookie)

Aufruf von Business Logik ohne zusätzlichen Schutz

```
http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243
```

XSRF Attacke per Social Engeneering

```
<a href="http://bit.ly/xyz">Link zu einer "vertrauenswürdigen" Seite</a>
```

XSRF Attacke per XSS

```
<img src="http://example.com/app/transferFunds?amount=1500&destination=attacker" />
```

XSRF

- Nur relevant wenn Session-ID oder Token per Cookie übertragen werden!

- Angreifer bringt Benutzer dazu URL aufzurufen
- Request wird mit Rechten des Benutzers ausgeführt
- Verschiedene Angriffsformen
  - Cross-Site-Scripting
  - Social-Engeneering / Unterschieben einer URL

- Cookies allein sind nicht sicher
  - Für Session-Cookie immer `httpOnly` und `secure` verwenden
  - Cookie kann nicht abgegriffen werden (per JS)
  - Cookie wird aber immer gesendet (XSRF immer noch möglich)
- Zusätzlicher Schutz notwendig

GEGENMASSNAHMEN

- Server
  - Schickt bei Login Session-ID als Cookie mit `httpOnly` und `secure`
  - Schickt bei Login zusätzliches Token als Cookie `XSRF-Token` ohne `httpOnly`
- Client
  - XSRF-Token wird zwischengespeichert (JS Variable) und Cookie gelöscht
  - XSRF-Token wird bei jedem Request als Header mitgesendet
- Server validiert bei jedem Request mitgesendetes XSRF-Token

ANGULARJS

- HTTP-Interceptor Konzept
- Interceptor schon mit dabei
  - Ließt Cookie `XSRF-TOKEN`
  - Sendet Header `X-XSRF-TOKEN`
  - Namen konfigurierbar
- Problem: *Öffne Link in neuem Tab*
- Lösung: Server sendet Token noch mal bei *GET api/login*

A8

XSRF + JSON

BEISPIEL

- Ausgangsituation
  - Benutzer in App eingeloggt (hat gültiges Session-Cookie)
  - Anwendung ist nicht oder unzureichend gegen XSRF geschützt

Aufruf der API ohne zusätzlichen Schutz

```
http://example.com/app/user
// returns ["Philipp", "secret"]
```

XSRF + JSON Attacke

```
<script type="text/javascript">
  var secrets;
  Array = function() { secrets = this; };
</script>
// load data via src (CORS)
<script src="http://example.com/app/user" type="text/javascript"></script>
<script type="text/javascript">
  alert('I stole your data: ' + JSON.stringify(secrets));
</script>
```

XSRF + JSON

- Funktioniert nur wenn XSRF möglich
- Funktioniert nur in wenigen Browsern
- Trotzdem absichern!

GEGENMASSNAHMEN

- Server: Prefixt JSON mit `)]}', \n`
- Client: Entfernt Prefix vor Deserialisieren
- Angular: Entfernt Prefix automatisch

ZEIT FÜR FRAGEN!?

- Philipp Burgmer
- burgmer@w11k.de
- @philippburgmer

- www.w11k.de
- www.thecodecampus.de
- @theCodeCampus